doi: 10.1515/acss-2014-0018
Applied Computer Systems
2014/16

State Synchronization Approaches in Web-based Applications

Aleksejs Grocevs[1], Natalya Prokofyeva[2], Stefan Leye[3],
[1–2]*Riga Technical University, Latvia,* [3]*Fraunhofer Institute for Factory Operation and Automation IFF, Germany*

***Abstract* – The main objective of the article is to provide insight into technologies and approaches available to maintain consistent state on both client and server sides. The article describes basic RIA application state persistence difficulties and offers approaches to overcoming such problems using asynchronous data transmission synchronization channels and other user-available browser abilities.**

***Keywords* – Persistence, state, synchronization, web application.**

I. Introduction

Nowadays many applications and tools are moving towards high-availability platforms, transferring services to a web environment. Such services, called RIA (Rich Internet Application), provide opportunities similar to the desktop-based solutions. Usually RIA consists of domain-specific environment and serves as a tool to ease required task performance on it. However, there are many Internet-specific problems that companies are trying to work out. One of these is application state synchronization across multiple application instances or even across single Internet browser restarts/page refreshes [1]–[3].

II. Problem Definition

Rich Internet applications usually contain many inputs, fields, drawings and other stateful elements, which can be altered or reversed to their original values by page reloading. Such alteration may cause changed information loss and will require to repeat all user actions leading to its exasperation and time waste. Similarly user-supplied data could be invalid or misconfigured; applications can incorrectly process those\these values, causing error notification and being unable to give a possibility to update and fix the data entered.

In client-side architecture (software development approach, where program that receives user input, e.g., browser, performs a lot of work and data preparation before sending it to a server, therefore reducing server workload and providing more responsive interface to an end-user), there are two places where data could be saved – either on a server remotely or on a client local computer. In case of multi-user environment, where multiple users work with the same information simultaneously, saving intermediate data on a client side could lead to data inconsistency and further partial data loss. Since that data should be merged with main information part on a server sooner or later, the sooner the data entered is uploaded to a central server-side storage, the better.

To synchronize data, which is already persisted on a client side with a server, browsers could use asynchronous calls like AJAX or web-workers, but if data loss is acceptable – you just wait until a user submits any form and attach the data required to form a request in hidden field(s). Multiple studies have already addressed this issue [4], the article does not only contain possible solution descriptions, but also provides comparison and recommendations suitable in particular cases.

Client-side storage type may vary based on RIA technology stack and data amount. The most common technologies for RIA development are HTML5 and Flex. Other plugin-based solutions like Java applets, JavaFX, Silverlight etc. are also in use, but much less frequently.

III. HTML and JS-based Application Data Persistence

Unlike previously mentioned technologies that require additional plugins and user interaction with the browser, JavaScript and HTML applications can work on virtually any modern browser. To persist some data locally, JavaScript provides several approaches to data retention between page refreshes.

A. Cookies

"Cookie" is the oldest local information storage way that has been accessible since the 1990s [5]. Modern browsers allow for multiple cookies per domain and usually these limits are 50 (Internet Explorer 8+) to 180 (Chrome 8+) or even 600 (Safari 5+) per browser session and from 4096 chars (IE, Safari) to unlimited per domain. This allows JavaScript to write and read data directly from cookies, which can be stored for a month in advance. Browsers typically have an option to overview stored cookies and an ability to remove unnecessary ones. To interact with cookie contents, browsers provide JavaScript document.cookie object, which contains all cookies set for current domain and all sub-domains [6]. To add a cookie, cookie string should be assigned directly to document.cookie variable, e.g.:

```
document.cookie = "foo=bar";      // (1)
document.cookie = "cb1Checked=true; max-age=" +
(60*60*24) + "; path=/login";     // (2)
document.cookie = "cb1Checked=; expires=Thu, 01 Jan
1970 00:00:00 GMT; path=/news";   // (3)
```

In the first example, a cookie named "foo" is created or replaced with a value of "bar". Since other arguments are omitted, a cookie will be automatically removed by a browser at the end of the current session. The "session" here is an

TABLE I

WEB STORAGE AND COOKIE LIMITS IN MODERN BROWSERS

Browser and version	**Web storage limit per domain, MB**	**Cookie size in, bytes**	**Cookie count per domain**
Chrome 29	5	4096	180
Firefox 24	5	4096	150
Safari 4	5	4096	No limit
Opera 12	3	5117	60
Internet Explorer 10	10	5117	50
BlackBerry Browser	25*	1024	10

* – Storage limit per origin (each [sub-]domain separately)

opened browser window, so after the browser restarts, a cookie will be erased.

In the second example, cookie "cb1Checked" is set to logical "true" with expiration time of 86400 seconds; hence, after 1 day the cookie becomes eligible for deletion. In addition, path argument is supplied, specifying that the cookie will be sent to a server only for pages under /login/* URI.

To remove a cookie earlier than the time limit specified in max-age or expires argument, the third assignment could be used. In this case, "cb1Checked" value is unset immediately (so its value is "undefined") and the whole cookie will be deleted after the next page refresh.

B. HTML5 Web Storage

Web storage in browsers gives larger than cookies place to store some data and to preserve it between page reloads. It is typically 5 to 25MB per domain, including subdomains. The comparison of overall default web storage and cookie data storage limits is given in Table I. There are two types of storage – localStorage and sessionStorage. As the name implies, sessionStorage can hold data only during a single browser session and after a browser restarts, its content is erased. If developers need to persist data for a longer period of time, localStorage could be used instead [7]. Both storages act like key-value data structures, similarly to an associative array. There are two main methods to retrieve and push objects inside storage, getItem() and setItem(), respectively. The following code demonstrates this approach:

```
var obj = localStorage.getItem("aboutText"); // (1)
sessionStorage.setItem("currentPage", 3);    // (2)
```

In the first example, JavaScript will try to fetch a value for "aboutText" key from local storage and assign it to variable "obj". If there were no object under such a key, the value of a variable would be "undefined".

In the second example, Number-type object will be written to temporary, session storage under "currentPage" key. If there were previous values – it would be lost and replaced with a new one.

Storage specification for HTML5 defines no particular type as value type for object persisting, but modern browsers are capable to handle only String type. That means that the following code will alert the text ""[object Object]"":

```
localStorage.setItem("testObj", {key1:"value1"});
alert(localStorage.getItem("testObj"));
```

Despite the fact that the written object contains at least one property called "key1", resuming the pushed value is not more than toString() call from the mentioned object.

To overcome this problem, it is possible to serialize objects before they get pushed inside Storage and deserialize when they are fetched from. The easiest way to transform JavaScript object into a text is performing JSON.stringify() method call. This object and method are available on modern browsers as well as Storage object itself. Thus, calling stringify() before setItem() and decoding JSoN object after retrieving it via getItem() and before real application usage (e.g., by calling eval() on the retrieved string. Fastest and less secure option) provides an opportunity to operate with regular JavaScript objects no matter what kind of object should be persisted.

Modern browsers have exception handlers and can handle irregular situations during JavaScript code run. Since these are runtime-type exceptions, unhandled exception (error) will halt further code execution. Regarding storage operations, developers usually implement QuotaExceededError handlers, i.e., surrounding potential dangerous calls with try/catch blocks:

```
try {
    sessionStorage.setItem("data", someBigObject);
} catch (e) {
  alert("No free space to persist user data");
}
```

In this scenario, a large object will be pushed to storage, but in case of overquoting an alert will be displayed to an end-user to notify about a problem. Cookies, unlike storage elements, would silently fail if total disk quota were exceeded.

Apart from larger data amount that can be persisted in storage, it is also possible to add event listeners for user-data saving and retrieving operations via set/getItem() calls. It is possible to bind to "storage" event type and execute an additional code for every storage changes that user-supplied data write calls could produce. This event will be fired in any other tab of same origin, excluding the original one, where storage update was called. This is useful for client-only data synchronization for applications that allow multiple invocations and want to synchronize data between instances without additional requests to a server.

C. IndexedDB

To overcome major storage pitfall – an impossibility to fetch ranged results – another HTML5 technology is under development – IndexedDB. Similar idea was implemented in Web SQL Database engine, but W3C ceased further specification development in 2010 [8].

IndexedDB provides asynchronous SQL-style queries to a local database that is embedded into a browser. Queries can also be wrapped inside transactions with rollback functionality and onError callback. The basic pattern to work with IndexedDB is as follows:

1. To initialize database connection and transaction;
2. To create a storage object (table);
3. To create a request;
4. To wait for request completion by using callback;
5. To process a request in callback.

It also possible to define indexes to improve retrieval performance and to additionally enforce some data type constraints, for example, by supplying {unique: true} index constraint.

Although IndexedDB could be used for user UI (user interface) state synchronization, it is impractical and too expensive to implement such a system as a client-side temporary storage engine.

IV. Flex- and Flash-based Application Data Storage

Usually Flex-based applications utilize persistent client-to-server connection and transmit objects and calls using AMF3 protocol [9], [10]. In this case, it is easier and faster to send all changes in UI immediately to a server rather than persisting them locally and waiting for an appropriate moment for synchronization. However, sometimes it is necessary to store user data from inputs or any other information in between application view transitions to restore that information back when it is required.

To store data locally on Flash client plugin, Adobe has implemented Local Storage, or Local shared objects. This storage restricted up to 100KB space in default configuration, where this restriction could be removed by a user and set to Infinity.

Storage objects are very similar to HTTP cookies. Shared objects are common for all Flash player instances in particular origin, its value is immediately transferred and applied to all instances as well.

To work with shared objects, SharedObject class is provided by ActionScript 3 implementation. To request a shared object from local storage, the following code can be used:

```
public var mySO:SharedObject;                    // (1)
mySO = SharedObject.getLocal("preferences");     // (2)
mySO.data.field1 = "value1";                     // (3)
mySO.flush();                                    // (4)
```

In the first line, the “mySO” variable of SharedObject type represents shared object instance that is retrieved from local storage or created in place if such a key in storage does not exist. To actually populate a variable value with object instance, there is call to fetch a local storage value for key “preferences” in the second line. On line three, the field updating process is shown, where a field value is indirectly assigned by using a “data” parameter. Lastly, shared object data in line four is flushed into persistent storage by calling flush() method on shared object instance. Specification states that explicit flush() call is not required though is desired to circumvent insufficient storage space prompt. If there is no free space left in storage, the method call returns SharedObjectFlushStatus. PENDING states and displays popup shown in Fig. 1.

If flush() method is not called immediately after data change, then the Flash plugin tries to update the shared object state in storage during plugin shutdown. This, however, can lead to inability to notify an end-user about possible free space issues due to plugin exit phase. Therefore, it is desirable to call flush() straight after the shared object data has been changed.

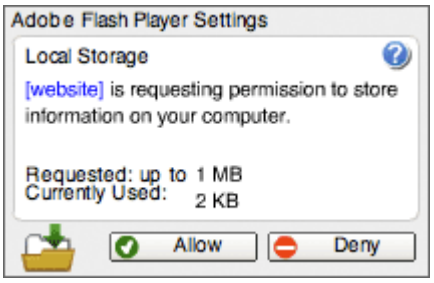

Fig. 1. Additional storage space requirements dialog shown by Adobe Flash plugin, providing information about current and requested storage data amounts.

Since storage space expansion dialog has 213 px in width and 136px in height, official guidelines suggests that SWF file is at least 215 pixels wide and at least 138 pixels high to fully show this modal dialog inside an application window.

V. Synchronization Handling Using Spring Web Flow

Spring Web Flow is designed to guide an end-user through a set of windows and prompts, providing an easy way to generate wizards, surveys, shopping cart check-out processes and other behavioral procedures that required a step-by-step solution with a predetermined step order [11].

Main request handling flow in Spring Web Flow starts within DispatcherServlet, which handles requests to all non-static files, such as images, CSS styles, JS scripts etc. DispatcherServlet transfers a request to FlowController, Spring MVC component which selects appropriate FlowExecutor to transfer the request to. FlowExecutor proceeds the request, handles state transitions and passes data further based on the information gathered from the request like supplied parameters and session data. According to the previously mentioned data, FlowExecutor selects the current flow and figures out in which state it currently is. FlowRegistry is the next component, being able to load, build and maintain flow definitions and is used to retrieve the corresponding flow data and state for FlowExecutor. To build the flow, FlowBuilderService is called next in sequence. Its

goal is to prepare all required services to process the current flow stage. UI/frontend elements are built together using ViewFactoryCreator (or usually MvcViewFactoryCreator if SpringMVC resolver is used) that is called from FlowBuilderService to generate final HTML page using ViewResolver mapper for physical resources, like JSP pages or another templating engine [12].

To enable Spring Web Flow in spring.xml, the following lines are required:

```
<flow:flow-executor id="flowExecutor" flow-
registry="flowRegistry"/>
<flow:flow-registry id="flowRegistry"
    flow-builder-services="flowBuilderServices">
    <flow:flow-location path="/WEB-INF/flow.xml"/>
</flow:flow-registry>
```

Both FlowExecutor and FlowRegistry elements are supplied, as well as FlowBuilderService processor class. To prepare flow definitions, flow.xml file should contain at least view state and end-state information for proper execution flow:

```
<view-state id="register"
view="account/registerForm">
    <transition on="submitReg" to="accountAdded"/>
    <transition on="cancelReg" to="cancelReg"/>
</view-state>

<end-state id="accountAdded"
view="externalRedirect:contextRelative:/home.do"/>
<end-state id="cancelReg"
view="externalRedirect:contextRelative:/home.do"/>
```

Since default Spring MVC InternalResourceViewResolver is used, view account/registerForm will be resolved as a physical file /WEB-INF/jsp/account/registerForm.jsp. The example above contains two transitions and two possible end states for a "register" flow. In both cases, these transitions lead to one end state – status code 301 HTTP redirect to home.do URI, located in context root. However, it is possible for transitions to lead to other states (e.g., views).

To access the created web-flow, a user must start with /register.do address relative to website root. After going through transitions, his state/position in flow will be persisted using Spring Web Flow internal request handlers. Such handlers parse POST-request form elements and based on ${flowExecutionUrl}, _eventId and other supplied parameters update user session information. Another approach to persist and therefore synchronize data between two browser requests is to utilize @SessionAttributes annotation in @Controller mappings in Spring MVC. To apply SessionAttributes, the following code can be used (surely getting information using DAO (data access objects) directly from user supplied data can compromise application security and preventive actions such as input validation/sanitizing should be taken on production environments):

```
@Controller
@SessionAttributes("cart")
public class CheckoutWizard {
    @RequestMapping("/step1")
    public String
checkoutForm(@RequestParam("cartId") int cartId,
ModelMap model) {
        ShoppingCart cart = cartDao.findOne(cartId);
        model.addAttribute("cart", cart);
        return "checkoutForm";
    }
}
```

In this example, all further steps will contain ShoppingCart object as a session attribute and cartId argument will be redundant. After saving object instance into model attribute "cart", RequestMapping-annotated method returns a mapping name, which will be resolved by ViewResolver, whose values could be populated from previously in-session saved state.

VI. Component State Persistence in Vaadin

Vaadin is a popular Rich Internet Application framework, which is capable to synchronize user changes in application in real-time using AJAX background synchronization. In version 7.1, an asynchronous push was added to reduce network usage for such requests. Vaadin provides many components with empty onChange() methods that could be overridden and called upon whenever element value or state is changed to synchronize current UI behavior and data representation on a client side with main repository data version on a server side [13].

For example, the most used HTML element for text input is <input>-type tag of type "text". To construct this element in Vaadin, the following code can be used:

```
PasswordField sample = new PasswordField();
sample.setImmediate(true);
sample.addTextChangeListener(new
TextChangeListener() {
    @Override
    public void textChange(TextChangeEvent event) {
        persistPassword(event.getText().length());
    }
});
```

Controls with "Immediate" flag will fire their *change() events immediately after focus loss. Calling persistPassword() method allows application to save temporary data about the changed but not yet confirmed field value in session or any other suitable storage. Additional types for synchronization that can be set by setTextChangeEventMode() are as follows:

- TextChangeEventMode.LAZY – An event is triggered when there is a pause in editing text. Pause length can be set by setInputEventTimeout() call. Similarly to TIMEOUT event mode, a text change event is forced to be fired before ValueChangeEvent, regardless of user character input flow;
- TextChangeEventMode.TIMEOUT – changes will be delivered to server only after the specified time;
- TextChangeEventMode.EAGER – event is triggered immediately after keypress or any other visual component change. Since a synchronization process is asynchronous, a user can continue typing without interruption.

Using Vaadin there is no need to implement any additional checks or logics for state synchronization on a client side [14]. If it is required, some of early mentioned client-side storages such as Web Storage or cookies could be used as well to provide redundant entered data persistence. That, however, would lead to additional data consistency problems but in case of Vaadin all data will be persisted in the order it arrives to a server. It is a particular project and a developer's task is to implement more sophisticated conflict resolution algorithms to avoid entered data collisions.

VII. Conclusion

Technologies and approaches described in the article provide information about possible data preservation and component/data synchronization opportunities between multiple application reloads or across multiple application instances. It is also possible to use multiple techniques at once, for example Vaadin for UI rendering/data manipulation and additional Flash shared objects as temporary local storage in case of the Internet connection loss.

Cookies, Web Storage and IndexedDB are always available for developers in modern browsers, so there is no need to program additional support for the mentioned technologies. If compatibility is a must – cookies should be used, since older browsers lack Web Storage and IndexedDB support.

On the other hand, if it is more preferable to implement synchronization as the part of existing web-solution, the mentioned modern frameworks, Spring and Vaadin, are the best choice since synchronization issues are handled internally by framework, leaving only business tasks to be solved by a developer.

By comparing local storage facilities, it has become clear that old cookie technology is not capable to deal with long text persisting and can be used only as a small character sequence holder like HTTP session key or currently selected language at a website. Besides, all cookies are sent to a server in every request, so transferred data amount would appropriately increase.

References

[1] A. Bongio *et al*., *Designing data-intensive Web applications*, Massachusetts: Morgan Kaufmann Publishers, 2003

[2] S. Salva, I. Rabhi, "Stateful Web Service Robustness" in *ICIW, 2010 Fifth International Conference*, 2010, pp. 167-173

[3] S. K. Beck, "Systems and methods for suspending and resuming of a stateful web application," U. S. Patent 7757239 B2, August 29, 2005.

[4] F. Bellucci *et al.*, "Engineering JavaScript state persistence of web applications migrating across multiple devices" in *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*, 2011, pp. 105-110.

[5] IETF. (1996, May) *Hypertext Transfer Protocol -- HTTP/1.0*. [Online]. Available: http://tools.ietf.org/html/rfc1945.

[6] Ianixt. (2011, March) *Browser Cookie Limits* [Online]. Available: http://browsercookielimits.x64.me/

[7] M. Mahemoff, (2010, October) *Client-Side Storage - HTML5 Rocks* [Online] Available: http:// html5rocks.com/en/tutorials/offline/storage/.

[8] World Wide Web Consortium, (2010, November) *Web SQL Database* [Online] Available: http://dev.w3.org/html5/webdatabase/

[9] C. Cao, J. Luo, Z. Qiu, "Technology of Application System Integration Based on RIA" in *International Conference CSIE 2011*, Zhengzhou, China, May 21-22, 2011. Proceedings, Part I: Springer Berlin Heidelberg, 2011, pp. 55-60.

[10] M. Ayenson, *et al.*, (2011, July) *Flash Cookies and Privacy II: Now with HTML5 and ETag Respawning*. [Online] Available: http://ssrn.com/ abstract=1898390

[11] D. Maciej, W. Zabierowski. "Web-based content management system," in *Modern Problems of Radio Engineering, Telecommunications and Computer Science 2010 International Conference*. June 2010.

[12] Willie Wheeler, (May 2008) *Build a Shopping Cart With Spring Web Flow 2.* [Online]. Available: http://springinpractice.com/2008/ 05/06/build-a-shopping-cart-with-spring-web-flow-2-part-2

[13] E. Duarte, L. Alejandro. *Vaadin 7 UI Design by Example: Beginner's Guide*. Packt Publishing Ltd, 2013.

[14] J. L.Williams, *Learning html5 game programming: A hands-on guide to building online games using Canvas, SVG, and WebGL*. Addison-Wesley Professional, 2012.

Aleksejs Grocevs. Master of science and engineering (2010). Primary field of study – high load web applications and modern technologies and techniques for high load optimization.

He currently works at Ambergames, Riga as the Head of Research and Development Department. Main research trends – internet-based application data transmission optimization.

Address: Meža Str. 1/4 – 533, Riga; Phone: +37126803626;
Fax: +37167089571; E-mail: aleksejs.grocevs@rtu.lv

Natalya Prokofyeva. Dr. sc. ing. (2007). Position: Riga Technical University, Computer Science and Information Technology Department, Chair of Software Engineering, an Associate Professor.

The main field of research activities: E-learning systems (model, methods, technologies), modern Internet technologies.

Address: Meža iela 1/4 – 533, Riga; Phone: +37129729846;
Fax.: +37167089571; E-mail: natalija.prokofjeva@rtu.lv

Stefan Leye. Diploma in Mechanical Engineering (2011). Primary field of study – integrated product development.

He currently works at the Fraunhofer Institute for Factory Operation and Automation IFF, Magdeburg – Germany, as a Project Manager of the Business Unit Virtual Interactive Training. Main field of research: Virtual reality and digital engineering solutions for product development and training.

Address: Sandtorstraße 22, 39106 Magdeburg, Germany;
Phone: +49 391 4090 114; Fax: +49 391 4090 115;
E-mail: stefan.leye@iff.fraunhofer.de