Engineering JavaScript State Persistence of Web Applications Migrating across Multiple Devices

Federico Bellucci, Giuseppe Ghiani, Fabio Paternò, Carmen Santoro
CNR-ISTI, HIIS Laboratory
Via Moruzzi 1, 56124 Pisa, Italy
{federico.bellucci, giuseppe.ghiani, fabio.paterno, carmen.santoro}@isti.cnr.it

ABSTRACT
Ubiquitous environments call for user interfaces able to migrate across various types of devices while preserving task continuity. One fundamental issue in migratory user interfaces is how to preserve the state while moving from one device to another. In this paper we present a solution for the interactive part of Web applications. In particular, we focus on the most problematic part, which is maintaining the JavaScript state. We also describe an example application to illustrate the support provided by our migration platform.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

Keywords: Migratory User Interfaces, Multi-device Environments, User Interface Adaptation, Continuity.

General Terms: Design, Experimentation, Human Factors.

INTRODUCTION
Recent advances in the capability of digital devices together with their progressive mass market penetration has led users to expect to be able to carry out their tasks in any context and in a seamless way regardless of the possibly changing settings.

In order to address this kind of challenging scenario, we propose our approach for migratory interactive applications, which are applications that are able to preserve the state reached after some user interactions using a specific device, and then resume such state within a new version of the application that has been migrated to the new device. The proposed architecture for migratory user interfaces is composed of a number of software modules, which support the dynamic generation of user interfaces adapted to various types of target devices and implementation languages, with the state updated to the one that was created through the source device.

The range of opportunities that migratory applications open up can be beneficial in radically different application domains: for instance, applications whose tasks require time to be carried out (such as games, business applications) or applications that have some rigid deadline and thus need to be completed wherever the user is (e.g.: online auctions). We focus on Web applications, which have limited support for state persistence and continuity across various types of devices. Thus, if for some reason users have to change device, the information entered can be lost and then they have to start over their interactive session on the new device from scratch.

Previous solutions for supporting migration [1] proposed techniques for the migration of entire applications, but this does not usually work because of the different interaction resources of the various devices. Kozuch and Satyanarayanan [5] proposed a migration solution based on the encapsulation of the volatile execution state of a virtual machine, but only limited to migration of applications among desktop or laptop systems. Chung and Dewan [2] proposed that, when migration is triggered, the environment starts a fresh copy of the application process in the target system, and replays the saved sequence of input events to the copy. However, this solution can have performance issues if such a sequence is long. Quan et al. [7] proposed to collect user parameters into an object called user interface *continuation*. Programs can create UI continuations by specifying what information has to be collected from the user and supplying a callback (i.e., a continuation) to be notified with the collected information. However, differently from them, we support the possibility of pausing the performance of a task, and then afterwards being able to resume the performance on a new device from the point the user left off. A toolkit for Distributed User Interfaces was proposed in [6], though our solution differs in that Web applications can be migrated without posing any constraint on the authoring technique to use for developing the applications. Other solutions for migratory interfaces [4] were able to manage only the state of forms and their adaptation process was not able to manage the associated scripts. In this work, we present a solution able to preserve the state during a Web migration, including not only the input entered by the users through the various interaction techniques available in the Web page, but also the state referred by JavaScript code (including Ajax scripts). In particular, the latter point represents the main contribution of this paper, since it has not been addressed in previous work on migratory user interfaces.

MIGRATION SOFTWARE ARCHITECTURE

Our solution is mainly a server-based approach. We did not implement our solution as a browser extension because our idea was to be as general as possible thus allowing users to freely choose whatever browser they like. In our solution we just suppose the existence of the desktop version of the page to be migrated. Also, we do not consider a migration occurring between two existing different versions of the application (e.g. migration from an existing desktop version to an existing mobile version of the same application). Rather, we judged more challenging and interesting to migrate such desktop version by means of dynamically building a new version suitable for the target device.

The proposed software architecture is illustrated in Figure 1. First, there is a device discovery phase (1), which allows the various devices available in the environment to discover each other. This is done through a communication between the Migration Server and the so-called "Migration Control Panel", a Web application (implemented in HTML and JavaScript) running on each migration device and allowing the user to manage the various migration features. In order to support the device discovery phase, the Migration Control Panel periodically announces itself (via Ajax requests) to the Migration Server, and then gets the list of available target devices. After this discovery step, and supposing a desktop-to-mobile migration, every time the browser currently running on the desktop requests a page via the Migration Control Panel (2), this request is captured by the Migration Proxy of the Migration Platform (3), which calls up the page concerned from the application server (4). Before sending the page back to the client device, the Migration Proxy annotates the page.

Annotations consist in modifying the accessed page in order to enable its migration. In particular, it includes i) adapting the links included in the page (to route any following connection through the Migration Proxy so as to support migration of pages that are accessible from the currently visited page), ii) adding IDs to the page components which can potentially be subject to migration (e.g. "DIV", "TABLE", "FORM"); iii) including appropriate JavaScript code in the original Web page so as to support the various migration features (e.g. capture and transmission of the current state of the page).

The Migration Control Panel also enables the user to trigger the page migration (5). When a migration trigger occurs, such a trigger has the effect of "waking up" a script method previously included in the original Web page in the annotation phase. Such a script method sends the DOM of the source page together with the current application state to the Migration Server (6). The communication between Migration Control Panel and the Web page is possible because the Migration Control Panel window keeps a reference to the Web page window, and thus can access data and structures of the Web application, which arrive via the Migration Proxy.

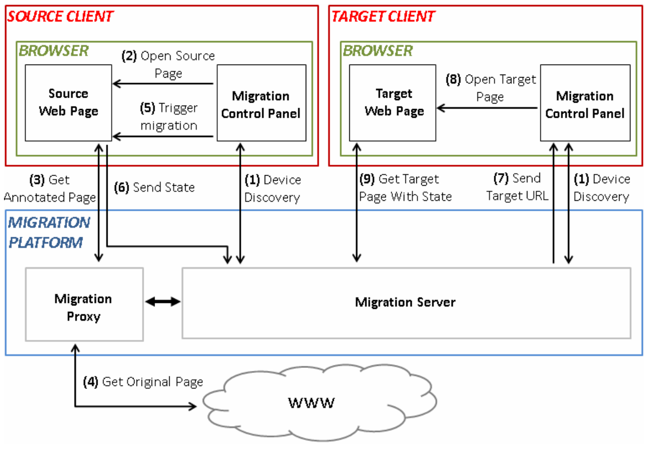

Figure 1. An overview of the migration architecture.

Once the Migration server has obtained the information about the current context in which the interaction is taking place (document, application state, focus), it generates the page for the new device with a state consistent with that of the original page, and sends its URL to the Migration Control Panel of the target device (7), which opens it in a new window (8). The new window shows the target page with state persistence obtained from the Migration Server (9).

The process which supports the generation of the page for the target device starting from the source device page is actually divided into a number of steps carried out by the Migration Server. First, by getting the DOM of the current page (which provides a description of the Web page considered) and the state of the page (namely: values contained within forms, currently selected options, etc.), the Migration Server returns as output a new page which is the original one enriched with the state information received in input (therefore, in the new page the form fields contain the updated values, etc.). Then, such resulting page undergoes a phase of reverse engineering, which builds the corresponding logical description from (X)HTML, CSS and JavaScript implementations. It is worth mentioning that when the Web application contains Flash or Java applets, then they are either replaced with alternative content provided by the application developers or they are passed "as is" to the target device, if it is able to execute them. The output of this reverse engineering phase (which is a concrete UI description for the desktop platform) is then transformed to a corresponding concrete UI description for the target platform, by mapping concrete interface elements on the source device into ones that are more suitable for the interaction resources available in the target device. Afterwards, the Migration Server analyses such a target logical description containing all the various presentations and identifies the currently focused presentation. Finally, the identified logical presentation is then transformed in order to build the corresponding implementation for the target device, which is sent to the target client so that the user can load the adapted page with the state resulting from the interactions occurred on the source device and then continue the interaction with the new version of the page.

The state that we preserve is composed of the values associated with all the forms elements, the current focus, the cookies, and the state of the JavaScript code. The last one has shown to be the most problematic aspect, which was not supported by previous solutions for migratory user interfaces and, thus, in this paper we discuss extensively its solution, whose importance also derives from the increasing use of JavaScript code.

MANAGING JAVASCRIPT STATE WITHIN MIGRATION

The problem of correctly managing the JavaScript state in migratory Web applications is a critical point, and since Web applications are becoming more and more interactive, it is likely to play an even more important role in the future. Indeed, if the state associated with JavaScript variables is not properly saved and restored, inconsistencies can be experienced when the user migrates to the target device. This means that exceptions could be raised due to the fact that some variables no longer exist in the new version uploaded on the target device, or even worse, no exception is raised, but some variables might hold incorrect values (namely, ones that are different from those held when the migration was triggered).

To capture and restore the JavaScript state of a Web application we basically use JavaScript code (automatically included in the concerned Web page by the Migration platform). Regarding the format for saving the state, we use the JavaScript Object Notation (JSON), since it is a lightweight format and in addition the JSON serialisation/parsing support is natively integrated within most currently available browsers.

The data types that are supported by the standard JSON format are: i) primitive types (Number, String, Boolean, null); ii) arrays (like [value1, value2, ...]); iii) associative arrays (also known as "Maps"), like {key1: value1, key2: value2, ...}. However, just using a standard JSON serialiser is not sufficient, since – as we will see – some problems are not appropriately handled by using it alone (object references, non-numeric properties of arrays, timers, …). In the following sections we describe the main issues we have addressed regarding the capture and restoration of migration JavaScript state, and the associated solutions we adopted. As we will see, such solutions include using a JavaScript library (*dojox.json.ref* [3]), which we have customized in order to properly handle specific issues. More specifically, we made a number of modifications to this library in order to serialise the objects that are not handled by standard JSON (Dates, array properties, DOM elements) and we manage the serialisation of objects that do not appear in global variables (e.g. timers), by using a library-independent mechanism explained in the following section.

Global and BOM variables

In JavaScript code, every object/variable defined in the global environment is simply a property of the global Window object (which in turn represents the browser window). In order to programmatically capture the values of the user-defined *global* variables, we use the JavaScript *for...in* statement, which enumerates the properties of objects (without knowing their names in advance). However, there are some window properties that, though enumerated, should not finally be included in the migration state. These are the properties belonging to the so-called Browser Object Model (BOM), an interface provided by the browser, which makes available a number of "utility" properties (e.g. the address of the page currently loaded in the browser, the reference to the DOM root, the history produced by using "Back" and "Forward" browser buttons). Such properties should be excluded from the migration state since on the one hand some of them are *browser-dependent*, while, on the other, some properties (like the reference to the DOM root) are already handled by the migration platform, thus they are useless for migration purposes. The mechanism we use to exclude the BOM-related variables is to create a "filter" list for each browser considered (Internet Explorer, Safari, Opera, Firefox, Chrome). This support works with any web application.

Object References

This case refers to when the migration platform has to serialise two variables or properties that refer to the same value and the value type is non-primitive (then, it is a type different from Number, Boolean, etc.). For instance:

```
var x= <anObject> ;
var y = x ;
```

In this situation, standard JSON would i) serialise twice <anObject> and ii) serialise it into two separate objects (one for the x variable and the other one for y). Instead, in our solution the result of the serialiser is that the y variable is associated with a unique *reference* to the x variable. With this mechanism (called *object referencing*) we avoid duplication of value serialisation and preserve, after migration, the fact that the *y* variable will continue to refer to the *x* variable.

Circular References

A special case of object references is represented by *circular references*. We have circular references when there is a variable, which is defined through another variable, which in turn is defined through the first one. Let us consider the following JavaScript excerpt:

```
var johnJohnson =  {
       name : "John" ,
        father : {
          name : "Paul" ,
          son : johnJohnson
        }
};
```

By using standard JSON, the variable *johnJohnson* could not be correctly serialised, since standard JSON serialises every object through its value and therefore an endless recursion will result, eventually raising an exception. In our case, in order to correctly preserve the object state, we serialise the *reference* to the object (not the value). This has been done by exploiting the *dojox.json.ref* library, which in correspondence of the value of the "son" property puts *a reference* by using a *path-based referencing* mechanism. The latter technique supports the identification of an object property by specifying its location within the object's structure. Thus, in this case, in order to identify the "son" property of the object we provide the *path* that goes from the root of the tree (where the tree represents the object), to the leaf (representing the property involved).

Timers

Timers are generally used when the developer wants to include some time dependency within the code (e.g. indicating that a certain portion of the code should be executed after a specific number of seconds). They are handled through the methods *setTimeout* and *setInterval* (resp.: to activate a single timer which triggers a handler at its end; to repeat a portion of code after a specific time interval); and through the *clearTimeout* and *clearInterval* methods (resp.: to stop a currently active timer, which is identified by an ID; and to clear a timer set with the *setInterval* method). In the following example *handler* is the code excerpt to run after *ms* milliseconds have elapsed; *timerId* is the identifier associated with the timer:

```
var timerId = setTimeout (handler , ms);
clearTimeout (timerId);
```

Timers can affect the state in two possible manners: first, we can have timers that are currently active/pending at the time when the migration occurs; secondly, in the code we might have variables containing references to timers. Unfortunately, the ECMAScript APIs neither offer methods enabling to access the state of an active JavaScript Timer, nor do they allow enumerating the list of active Timers.

In order to cope with this issue and allow the correct restoring of timers after migration, in our solution we override the standard behaviour of *setTimeout* by adding additional code able to appropriately handle the state of timers. More specifically, the solution we adopted creates a global/public list of Timers, by adding a Timer object to such a list every time the *setTimeout* (or *setInterval*) is invoked. Since such a list is global, it will be easily accessed in the global state. The timers of such a list will be updated by a single "central" timer which will invoke at regular intervals the update function of each active timer. In order to correctly restore timers after migration, in the target device we build a list of timers starting from the global/public list of timers saved in the state and then, by re-starting the central timer, all the connected timers will also be re-started. In this way, the active timers will be consequently restored in the target device.

Dates

The Date object is used to work with dates and times. Standard JSON does not support the serialisation of a Date object since it serialises it as a *void* object. The *dojox.json.ref* library provides only a partial solution for this problem. Indeed, it correctly serialises the Date object into an ISO-UTC –formatted string, but it is not able to re-convert it again into a Date object without explicitly instructing the deserialiser with a list of Date property names. However, even doing this, the solution was not able to cover some situations, for instance when an object has multiple properties of Date type at different nesting levels and the same name. In this case only the first occurrence was correctly preserved with that technique. Our solution is simpler and more general: we encapsulate the ISO-formatted date in an object holding a property that marks the object in such a way that the deserialiser can quickly identify and correctly restore the object as a Date Object.

Properties dynamically assigned to objects

In JavaScript all the non-primitive data types (like associative arrays, arrays, strings, functions...) are objects and, as such, can have *dynamically assigned* properties. Such properties are saved in the state by standard JSON only if the object containing them is an *associative array* (or *Map*), otherwise (namely, if they are included by other types of objects) they are ignored. An example of this problem can occur while serialising the following excerpt:

```
var array = [value1 , value2];
array.dynProp = someValue;
```

In order to manage this issue, we have identified a solution that successfully manages saving such dynamic properties also for other types of objects apart from associative arrays. So far we have implemented this solution for managing arrays, though our solution is easily extendible to other cases. In our solution we appropriately modified the *dojox.json.ref* serialiser in such a way that it encapsulates each array in an object holding a property that contains all the array's dynamic properties.

References to DOM nodes

The JavaScript of a web page can access the DOM tree nodes by reference (e.g. to read or modify them). Thus, the persistence of such references is needed in order to preserve the state.

Indeed, one of the most common operations carried out on the DOM tree is to find a node by providing its identifier (by using the *getElementById* method, which provides a reference to a DOM node that has a unique ID). Unfortunately, standard JSON is not able to correctly serialise JavaScript variables containing references to DOM nodes. The following cases can happen: either the

concerned element has an ID, or it does not have an ID, or it does not belong to the DOM.
Consider the following excerpt:

```
var element = getElementById("myHtmlElement");
var image = new Image("imageSource");
```

In our solution, when an object of type HtmlElement is actually a reference to a DOM node, we verify whether it has an ID. In this case, such an ID is saved within the state, otherwise our solution assigns an ID to it. In the remaining case (when the object does not belong to the DOM, like the *image* variable included in the above code excerpt), the value of the object is saved by storing the values of all of its public properties using the JavaScript library JsonML (http://jsonml.org/).

SAVING, TRANSMITTING AND RESTORING THE STATE WITHIN THE MIGRATION PLATFORM

In this section we focus on how the migration platform supports the saving, transmission and restoring of the JavaScript state of Web pages. The core of our solution is represented by the JavaScript *JSStateMigrator* library we have developed for this goal. Such a library has two main methods:

- *saveState():* saves the current JavaScript state into a JSON message represented by a Map object, and includes all the global variables of the application and their properties;
- *loadState()*, takes as input parameter the JSON message representing the state of the migrated application and restores it within the target device.

In the following sections we describe them further.

Saving the State

The *saveState()* method works in the following way. As previously mentioned, we use the *for…in* statement as the main mechanism for saving the value of all the JavaScript global variables (which are properties of the window object). As also noted before, we added to the *for...in* cycle a condition to check whether a certain property has to be excluded since it belongs to the BOM variables. After having done this, we create a JSON message containing a couple (key, value) for each property of the concerned global object. It is worth noting that the serialised JavaScript state is no longer in standard JSON, because the serialised properties are enriched with additional information that will enable the custom deserialiser to correctly handle special objects that cannot be [de]serialised by using standard JSON (e.g. Date objects, array properties with non-numerical keys, DOM elements, or even HTML elements which are not in the DOM). However, the entire JavaScript state is now included in such a JSON message.

Transmitting and restoring the state

Regarding the transmission of such a JSON message, this is carried out by an AJAX request directed to the Migration Server. Through such a request, the client passes both the DOM of the page and the JavaScript state to a servlet (which is on the Migration Server). According to such information, the servlet creates the corresponding page. This means that first the servlet identifies the <body> element of the page derived from the received DOM. Then, it appends a JavaScript function, whose goal is to update the JavaScript variables contained in the page, to the current content (if any) of the *onload* attribute of the <body> element. After having done such modifications to the page, the servlet stores the updated page (containing the up-to-date state) in a specific location of the server, and also stores the JSON message received from the AJAX script in a file. According to such information, the server then builds the URL from which the target device browser will load the page with the updated state. So, by analysing the received JSON message the target device browser will be able to restore the state in the target device (*loadState()* method). This is done by deserialising all the properties of the abovementioned JSON object and restoring them onto the corresponding global variables. Then, the pending timers are also created and the central timer is started again (as explained above). So, the actual restoring of the JavaScript state of the Web application is carried out within the client, after the Web page has been loaded.

AN APPLICATION EXAMPLE: JS-TETRIS

To show an example application, we consider a Web (XHTML+JavaScript) implementation of Tetris (http://www.gosu.pl/tetris/), a widely known arcade game. The considered game (see Figure 2) is composed of a 12x22 grid, in which objects with different geometrical shapes fall to the bottom of the game board. The player controls the pieces by moving/rotating them, trying to make them fit each other so as to compose a horizontal line through them. When this occurs, the line(s) of pieces disappear and the score increases. As the horizontal lines are completed, the game becomes more difficult (e.g. the pieces fall progressively faster). Alternatively, if the player leaves holes within the horizontal piece rows, they do not disappear but start to pile up, until the player is no longer able to play, and the game will end. The UI of the game is composed of a number of nested DIV elements. First, there is the top level DIV (representing the whole game), which in turn is vertically decomposed into a DIV element (for the menu) and another DIV element (with the game board).

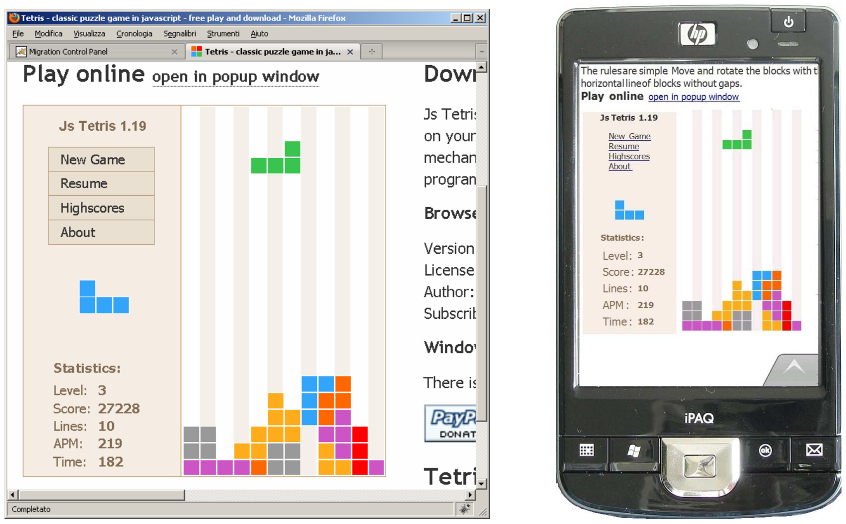
Tetris - classic puzzle game in javascript - free play and download - Mozilla Firefox
Play online open in popup window
Js Tetris 1.19
New Game
Resume
Highscores
About
Statistics:
Level: 3
Score: 27228
Lines: 10
APM: 219
Time: 182
The rules are simple Move and rotate the blocks with t
horizontal line of blocks without gaps.
iPAQ

Figure 2: Desktop-to-mobile Migration of the Example

The menu is in turn composed of a number of DIV elements, each of them including a menu item (the control buttons, the next piece, the score data, ...). The board area is a DIV element which initially contains only the twelve vertical columns, which act as guides for the pieces. However, as soon as the game progresses, such elements will contain the various small, elementary squares (each of them represented by a DIV element) composing each game piece. The game layout and the initial positioning of its elements are handled by an associated CSS stylesheet.

All the JavaScript code is included within the constructor function *Tetris*, instantiated after the Web page is loaded, and then saved in a global variable. Within this function, various nested functions are defined: *start*, *reset*, *pause*, *gameOver*, *random*, as well as *up, down, left, right, space* in order to control the pieces. In addition, other constructor functions are defined such as *Window*, *Keyboard*, *Area*, *Puzzle*, *Stats*, *HighScores*, which define corresponding properties of the game. In addition, in the Tetris code there is also the definition of some anonymous functions, which are assigned to properties of DOM nodes, for instance ($("tetris-menu-start").onclick is for starting a new game). Finally, there are also additional properties defining the board area (*unit*, *areaX*, *areaY*). In the desktop-to-mobile migration of the Tetris game (see Figure 2) our migration platform exploits some of the features for managing the JavaScript state we described before. Indeed, this game has object references such as references to HTML elements, and timers.

CONCLUSIONS AND FUTURE WORK

In this paper we have presented our approach for supporting JavaScript state persistence and, consequently, task continuity in interactive Web migratory applications, and describe its application to a case study in the game domain. Future work will be dedicated to carrying out a number of user tests to assess the usability of the presented solution on various case studies.

ACKNOWLEDGMENTS

We gratefully acknowledge support from the Artemis EU SMARCOS and the ICT EU SERENOA projects.

REFERENCES

1. Bharat, K. A. and Cardelli L. Migratory Applications. In proceedings of User Interface Software and Technology (UIST ‘95), 1995, pp. 133-142.
2. Chung, G., Dewan P. A mechanism for Supporting Client Migration in a Shared Window System, Proceedings UIST’96, pp.11-20, ACM Press.
3. Dojox.json.ref library, available at http://docs.dojocampus.org/dojox/json/ref
4. Ghiani, G., Paternò F., Santoro C. On-demand Cross-Device Interface Components Migration, Proceedings Mobile HCI 2010, pp. 299 – 308, 2010, ACM Press.
5. Kozuch, M., Satyanarayanan, M. Internet Suspend/Resume, Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA’02) IEEE Press, 2002.
6. Melchior, J., Grolaux, D., Vanderdonckt, J., Van Roy, P. A toolkit for peer-to-peer distributed user interfaces: concepts, implementation, and applications. Proceedings ACM EICS 2009: 69-78.
7. Quan, D., Huynh, D., Karger, D. R., and Miller, R. User interface Continuations. Proceedings 16th ACM UIST Symposium. 2003. ACM Press, pp. 145-148.