The Liquid.js Framework for Migrating and Cloning Stateful Web Components across Multiple Devices

Andrea Gallidabino
Faculty of Informatics, USI Lugano
via Buffi 13, 6900 Lugano
andrea.gallidabino@usi.ch

Cesare Pautasso
Faculty of Informatics, USI Lugano
via Buffi 13, 6900 Lugano
c.pautasso@ieee.org

ABSTRACT

We are heading toward an era in which users own more than one single Web-enabled device. These devices range from smart phones, tablets and personal computers to smart Web-enabled devices found in houses and cars. The access mechanisms and usage patterns of Web applications are changing accordingly, as users interact more and more with Web applications through all their devices, even if the majority of Web applications are not ready to offer a good user experience taking full advantage of multiple devices. In this demonstration we introduce Liquid.js, a framework whose goal is to enable Web developers to take advantage of multiple heterogeneous devices and offer to their users a liquid user experience, whereby any device can be used sequentially or concurrently with Web applications that can effortlessly roam from one device to another. This way, as highlighted in the demonstration users do not need to stop and resume their work on their Web application as they migrate and clone them across different devices. The demo will also show how developers can easily add such liquid behavior to any Polymer Web component.

Keywords

Web Components, Liquid Software, Liquid Web Applications, Stateful Web Components, Multi-Device Environments

1. INTRODUCTION

The term *Liquid Software* is derived from a metaphor: like a liquid adapts to the shape of its container, liquid software can adapt to take full advantage of every device it is deployed on [1]. A liquid Web application [8] is able to flow between multiple devices following the attention focus of its users [7].

The concept of liquid software was first introduced in 1999 in the context of active network research. Hartman et al. [6] tie the notion of liquid software to code mobility [3] and injected code. The methaphor has evolved over the years into the liquid software manifesto [10] to emphasize specific properties of the user experience. The manifesto takes into consideration the fact that nowadays end users own multiple devices and that they wish to use all of them together to run their applications.

As a first step towards achieving this vision, this demo presents the novel architecture and API of the Liquid.js framework. We describe its main abstractions (the Liquid component and Liquid variable) to share storage resources in order to transparently maintain the state of a liquid Web application synchronized.

WWW'16 Companion, April 11–15, 2016, Montréal, Québec, Canada.
ACM 978-1-4503-4144-8/16/04.
http://dx.doi.org/10.1145/2872518.2890538.

2. MOTIVATION

Nowadays users own two or more Web-enabled devices [5], usually a smart phone, a tablet and a personal computer. The design of the majority of the Web applications does not yet take into consideration the scenario in which users access applications with all the devices they possess [4]. Those devices may be used sequentially – where the work made on one devices needs to be transferred to the other as the user continues to use the same application there – or simultaneously. Traditional Web 1.0 applications, which do not use sessions, make it possible to simply transfer them across devices by sharing the link to the page currently open by the user. As soon as sessions are introduced (e.g., when the user needs to be authenticated) more effort is required to make the application flow between different Web browsers on multiple devices. Modern Web applications, which keep a significant amount of state on the client, make it even more complex for developers to achieve a liquid user experience as their architecture was not designed to withstand the interaction of the same user using multiple Web browsers on different devices, possibly at the same time.

The goal of the Liquid.js framework is to help developers build Web applications that can take full advantage of all devices owned by their users, what has been identified as the *Liquid User Experience* [8].

3. THE LIQUID.JS FRAMEWORK

The idea behind the Liquid.js framework is to give to developers the tools to easily create applications transparently running on multiple heterogeneous devices. There are three different use-cases of the Liquid.js framework: **sequential screening**: applications run on different devices at different times; **simultaneous screening**: applications run on different devices of the same user at the same time; **collaboration**: two or more users collaborate using the same application running on all of their devices at the same time.

The framework enables the transparent creation of distributed Web applications running on multiple devices, which need to operate on a shared decentralized state. The assumption is that these applications are developed using Web components, which provide the necessary level of granularity to structure the application user interface and its state. Liquid.js takes care of the state synchronization by using Yjs [9], a concurrency control and conflict resolution framework.

3.1 Liquid components

The Liquid component is a piece of mobile code [2] which encapsulates *behaviors* (JavaScript) and *UI* (HTML/CSS). Whenever a Liquid component is needed, it is dynamically loaded in the Web browser's currently open page through *HTML imports*.

We decided to build Liquid components on top of Web Components, specifically using the Polymer syntax. Thanks to the Web components standard the instances of a Liquid component are isolated from each other, making it possible to effectively componentize and build applications with multiple instances of the same component, while also making it possible to explicitly describe which part of the component state needs to be synchronized.

Figure 1 shows the architecture of the Liquid component. A component is defined by its *Application specific HTML and behavior* implemented by the developers using the Liquid.js framework API to manage the replication and synchronization of its state. The Liquid Component Library takes care of propagating and synchronizing changes of the state for each component instance. The developer of the component does not need to worry about how many instances are running or keep track of the set of devices that are connected to the application on behalf of the user. The communication with the other Liquid components is also managed transparently from the developer perspective. Our current implementation uses direct browser-to-browser connections through WebRTC Datachannels, if it is available on the Web browser, or through WebSockets relayed by a Web Server (called Liquid Server in Figure 1) otherwise.

In addition to establishing the WebRTC/WebSocket communication channel, the Liquid Server is used for device discovery and pairing as well as for storing the repository of Liquid components that will be downloaded and instantiated on the Web browsers used to run the liquid Web application.

3.2 Liquid variables

The State of a liquid Web application is decomposed into Liquid variables, which are associated to the data model of each Liquid components. A variable is identified by its name and stores its current value, which can be of any JSON-serializable JavaScript data type.

While each Liquid component instance always holds the current value of a Liquid variable, the Liquid.js framework may choose to replicate and manage the state outside the Liquid component. Values of Liquid variables are automatically synchronized among paired component instances, according to the permissions associated to each variable.

3.3 Permissions

The permissions of a variable affect how its state can be changed and propagated among paired components. There are two Boolean permissions: **publish** and **subscribe**. The publish permission defines if a component is allowed to propagate changes to the state of a variable to other paired component. The subscribe permission defines if a component is allowed to accept changes from another paired component.

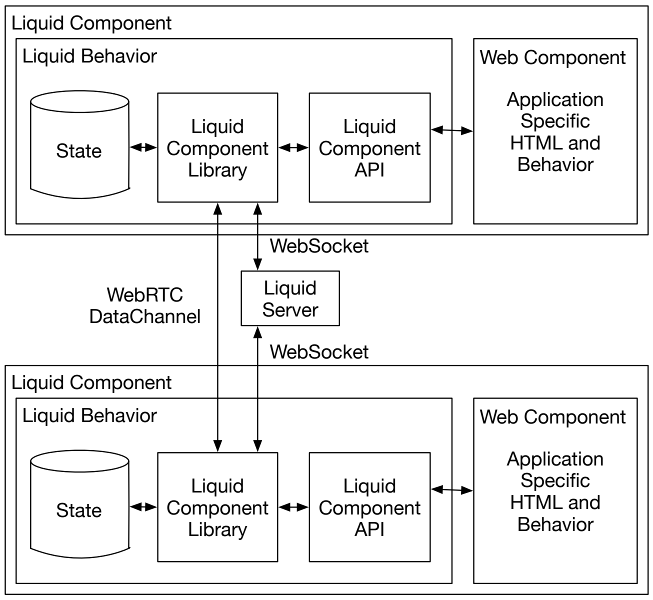

Figure 1: Liquid component architecture

4. DEMO

4.1 Liquid User Experience

We will show several Web applications built with the Liquid.js framework. They will show the effect of different sharing policies associated to the liquid variables (*global*, *shared*, and *local*) and the two permissions *publish* and *subscribe*. For example, Figure 2 shows one of the possible applications composed by three different Liquid components: *Presenter Screen* which shows in a text-area all the letters input by a keyboard; the *Keyboard* which is used to input letters; and the *Viewer Screen* which shows the same text shown inside the *Presenter Screen*. In this application *Presenter Screen* pairs itself with a *Keyboard* component. Since they both define an *Input Letter* variable they share it. The keyboard publishes the input letter in the presenter screen, which allows subscription to the keyboard. Similarly the *Presenter Screen* pairs with many *Viewer Screens*. The viewer screens allows subscription to the variable *Screen*, the presenter screen publishes any new value to all viewers.

The components can be scattered among multiple devices as the user pleases without compromising their state synchronization. Figure 3 shows two screenshots of the Keyboard component and the Presenter Screen component encapsulated in the liquid frame implemented in the current version of Liquid.js. The Liquid frame default implementation can be overridden. Not only the Liquid frame is optional, but the developer can completely redefine it by using the component deployment lifecycle API. In particular, we will demonstrate a push-based approach, where users interact with existing instances of a component to move them

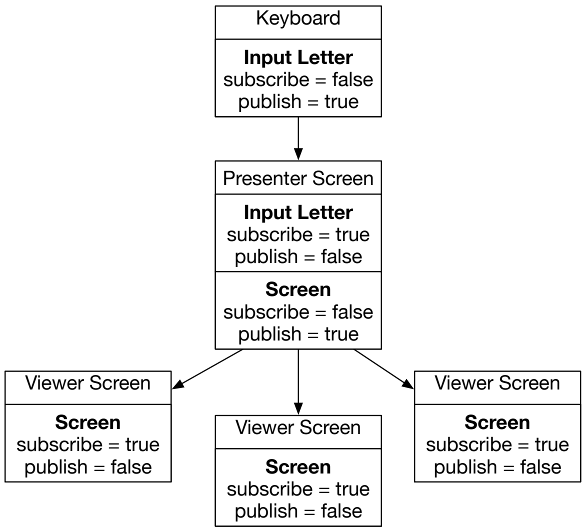

Figure 2: Presenter - Viewer application

elsewhere. We are also experimenting with pull-based approaches, where the target device is used to choose the component instance that should be migrated on it.

We will also demonstrate the liquid experience with real Web applications, such as: peer to peer picture sharing by grabbing frames from WebRTC video feeds; collaborative trip planning over a Google Map component, where the planned route can be configured on a desktop and then shared with a mobile device.

4.2 Liquid.js API

Liquid.js provides two APIs: the *Component Deployment Lifecycle* API, which provides the primitives to manage the creation, pairing and removal of Liquid components; and the *Liquid State Storage* API, which lets developers control how the state of the components is synchronized.

In particular we will show that whenever the Liquid.js library is loaded it automatically creates an unique identifier *deviceID*. This id identifies devices connected to the Liquid application. The *Component Deployment Lifecycle* API exposes the following four primitive methods:

- **create(componentType [, deviceID] [, state])**:
  The *create* method creates a new instance of a component and returns its URI identifier *componentID*. The type of the component is chosen with the *componentType* parameter (e.g. *liquidKeyboard*, *liquidWebcam*). If the optional parameter *deviceID* is defined, the method will create the Liquid component on the specified device. In both cases, the component will be attached to the Web page DOM as a child of a container element which can be configured when Liquid.js is loaded. The default initial state of the newly created component can be overridden with the optional parameter *state*.
- **pair(componentID_1, componentID_2)**:
  The *pair* method makes *componentID_1* and *componentID_2* share the state of their liquid variables. As we are going to discuss, depending on the permissions associated with the variables, pairing can be symmetric (state changes flow in both directions) or asymmetric (one component will follow changes in the other one).
- **unpair(componentID_1, componentID_2)**:
  The *unpair* method stops the synchronization between the two components. The state of the components is not affected, but future changes will not be synchronized.
- **remove(componentID)**:
  The *remove* method removes a component instance from the device it is currently running on. The user will no longer see the component. If the component was paired with any other Liquid component, the other components are automatically unpaired with it.

The *Component Deployment Lifecycle* API also exposes the following utility methods (migrate, clone, and fork) which are a composition of the previously defined primitives.

- **migrate(component, deviceID)** =
  create(component.type, deviceID, component.state) + remove(component.ID)
- **clone(component, deviceID)** =
  create(component.id, deviceID, component.state) + pair (component.ID, createdComponent.ID)
- **fork(component, deviceID)** =
  create(component.type, deviceID, component.state)

The second API provided is called *Liquid State Storage API*. The Liquid State Storage API is used during liquid components development in order to declare which state variables should be managed by Liquid.js so that their values get automatically synchronized. The API exposes the following three methods that are accessible from inside a Liquid component:

- **registerVariable(name, init[, permissions])**
  The *registerVariable* method registers a variable as Liquid. This method notifies the Liquid behavior inside the Liquid component that a variable called *name* should me monitored and is initialised with value *init*. Whenever the component is paired with another component, this variable will be automatically synchronized following the *permissions* passed as parameter.
- **variableChange(name, value)**
  The *variableChange* method updates the value of the variable called *name*. Liquid.js propagates the change to all components sharing this variable.
- **unregisterVariable(name)**
  The *unregisterVariable* method unregisters variable *name* so that it will not be synchronized anymore with other paired components.

4.3 Liquid Web Component Development

Based on the previously shown APIs, we will show a step by step tutorial on how to liquefy any Polymer element in the Polymer element catalog[1]. Listing 1 shows the final code derived by the liquefaction of the Polymer *paper-input*[2] component. The demo will go through the following steps:

1. Create a new component.
2. Define a new template in the newly created component and insert the Polymer component (e.g., *paper-input*) we want to *liquefy*.

[1]`https://elements.polymer-project.org/`
[2]`https://elements.polymer-project.org/elements/paper-input`

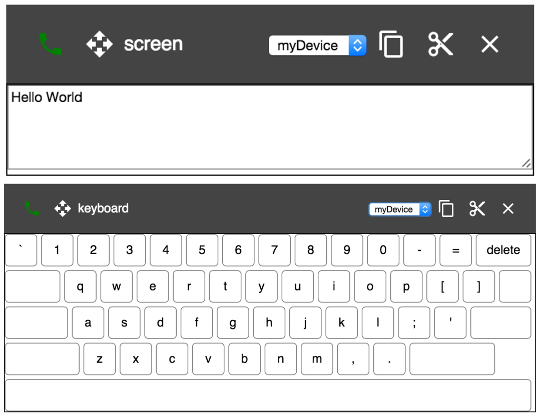

Figure 3: Paired Liquid Screen and Liquid Keyboard working together

3. Define a new property and bind it to the Polymer Component we chose in the previous step. Include an observer function in the property.
4. Register the new property as a Liquid variable by passing its name and a default value (and optionally its permissions) to the API method *registerVariables*.
5. Inside the observer function call the API method *variableChange* by passing the name of the variable and the changed value.
6. Run the liquid component and show that it can be dynamically deployed and migrated on different devices. When cloning the component its state gets automatically synchronized.

Listing 1: Liquid Polymer Paper-Input Component

```
<link rel="import" href="/liquid-behavior/
    liquid-behavior.html">
<link rel="import" href="/paper-input/
    paper-input.html">
<dom-module id="liquid-page-example">
 <template>
  <paper-input label="Insert Text Here"
      value="{{liquefyVariable}}"/>
 </template>
 <script>
  Polymer({
   is: 'liquid-page-example',
   behaviors: [LiquidBehavior],
   properties: { liquefyVariable: {
       observer: 'textChanged'}},

   attached: function() {
    this.registerVariables([{
     name:'liquefyVariable',
     init: 'Insert Text Here'
    }])
   },

   textChanged: function (newValue,
       oldValue) {
    this.variableChange('liquefyVariable',
        newValue)
   }
});
</script>
</dom-module>
```

5. CONCLUSION AND FUTURE WORK

While developers can take advantage of existing techniques to make their components responsive so that they can adapt and take advantage of the capabilities provided by heterogeneous devices [11], we are currently improving the support for automatic adaptation in the Liquid.js framework. We plan to further extend the liquid concept beyond the visible components of the user interface of a Web application. Currently, every device on which a component is deployed is running its behavior using local resources. Paired devices share the state of liquid components, but perform computations over it separately. This in some cases may lead to redundant computations being performed and unnecessary conflicts over the shared state. We are working to make it possible for non-visual components of liquid Web applications to transparently share the computational power of all connected devices.

6. REFERENCES

[1] D. Bonetta and C. Pautasso. An architectural style for liquid web services. In *Proc. of the 9th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 232–241, 2011.

[2] A. Carzaniga, G. P. Picco, and G. Vigna. Designing distributed applications with mobile code paradigms. In *Proceedings of the 19th international conference on Software engineering*, pages 22–32. ACM, 1997.

[3] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *Software Engineering, IEEE Transactions on*, 24(5):342–361, 1998.

[4] Google. The new multi-screen world: Understanding cross-platform consumer behavior. `http://services.google.com/fh/files/misc/multiscreenworld_final.pdf`, 2012.

[5] Google. The connected consumer. `http://www.google.com.sg/publicdata/explore?ds=dg8d1eetcqsb1_`, 2015.

[6] J. J. Hartman, P. Bigot, P. Bridges, B. Montz, R. Piltz, O. Spatscheck, T. Proebsting, L. L. Peterson, A. Bavier, et al. Joust: A platform for liquid software. *Computer*, 32(4):50–56, 1999.

[7] M. Levin. *Designing Multi-device Experiences: An Ecosystem Approach to User Experiences Across Devices*. O'Reilly, 2014.

[8] T. Mikkonen, K. Systä, and C. Pautasso. Towards liquid web applications. In *Proc. of ICWE*, pages 134–143. Springer, 2015.

[9] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma. Yjs: A framework for near real-time p2p shared editing on arbitrary data types. In *Proc. of ICWE*, pages 675–678. Springer, 2015.

[10] A. Taivalsaari, T. Mikkonen, and K. Systa. Liquid software manifesto: The era of multiple device ownership and its implications for software architecture. In *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*, pages 338–343. IEEE, 2014.

[11] J.-P. Voutilainen, J. Salonen, and T. Mikkonen. On the design of a responsive user interface for a multi-device web service. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*, pages 60–63, 2015.