Migrating and Pairing Recursive Stateful Components between Multiple Devices with Liquid.js for Polymer

Andrea Gallidabino

Faculty of Informatics, University of Lugano (USI), Switzerland
{name.surname}@usi.ch

Abstract. With the continuous development of new Web-enabled devices, we are heading toward an era in which users connect to the Web with multiple devices at the same time. Users expect Web applications to be able to flow between all the devices they own, however the majority of the current Web applications was not designed considering this use case scenario. As the number of devices owned by a user increases, we have to find new ways to give Web developers the tools to easily implement the expected liquid behaviour into their software. We present a new contribution provided by the Liquid.js for Polymer framework, allowing the migration of recursive component-based Web applications from a device to another. In this demo paper we will show how to create recursive components, how to migrate them among devices, and how their state can be paired among the various components.

Keywords: Web Components, Liquid Software, Liquid Web Applications, Stateful Web Components

1 Introduction

The Liquid.js for Polymer framework is based on the *Liquid software* paradigm [1, 2]. As a poured liquid adapts to the shape of the containers holding it, a software adapts to the resources of the devices running it [3]. Liquid applications are specifically designed to run on multiple devices following the users focus whenever the application flows among them [4]. From the user point of view these applications run either: 1. **sequentially**: at any given moment in time an application runs only on a single device, however the users may decide to move the application to a different one. The state of the application in a sequential scenario is never accessible from two devices simultaneously; 2. **simultaneously**: the application has to be shared among multiple devices, while its state has to be kept in sync.

Liquid.js for Polymer [5] is a novel framework for easily create liquid Web applications. In a previous demo publication we showed how to create liquid Web applications by using our framework API, in this demo we focus on a new feature: while in the past the migration was only possible on *flat* components,

today we make a distinction between *container* and *leaf* components. Finally we show how these complex structures can be paired with each other by invoking methods provided by our API.

2 Liquid Framework

Liquid.js for Polymer extensively exploits the most recent HTML5 standards. This approach allows to target as many devices as possible, achieving an increased compatibility with any system or hardware able to run a Web browsers complying with HTML5, like Google Chrome or Mozzilla Firefox.

The goal of Liquid is to automatise how an application is *shared* between multiple devices, the framework transparently decides where data has to be stored in such a way it is always available and as close as possible to the source using it [5]. The environment created by Liquid is highly decentralised, the decentralisation is achieved by delegating clients of storing data in a peer-to-peer mesh instead of storing it in a central server and, whenever possible, clients also distribute the application assets whenever they are requested. The server in a Liquid application is used as a fallback whenever P2P technologies are not available, and as the initial orchestrator of the P2P mesh by exchanging signalling messages between the clients.

Liquid expects the developer to be able to build component-based application by using the WebComponents standard, specifically by using the Polymer library[1]. A developer has to decompose an application into smaller components and he has to *explicitly* define which parts of the application are expected to be shared between devices. In order to do so Liquid provides an API and gives default tools to the Web developers for easily *migrate*, *fork* and *clone* stateful liquid Polymer components between multiple devices: – **migrate**: the migrate primitive moves a component from a device to another, the state of the component is migrate as the component does. No trace of the component and the state is kept on the initial device; – **fork**: Liquid makes a copy of a component and its current state on another device. The state of the initial component and the newly created one is not synchronised, meaning that upon state change they don't affect each other; – **clone**: Liquid makes a copy of a component and its current state on another device, while keeping the state of the two components automatically synchronised. These three behaviours can be imported into any Polymer component by adding our liquid behaviour to it, in the case a Polymer components import the liquid behaviour we call it a liquid component.

3 Demo

The demo will focus on a new feature of Liquid.js: *liquid container components*. While the liquid components, discussed in the previous session, only import the

[1] `https://www.polymer-project.org/1.0/`

three liquid primitives into a solid component, they do not allow the composition of multiple liquid components into one, which is an expected use case scenario whenever developers decide to use the WebComponents standard and the component-base architectural style. For this reason we introduce the concept of *container* and *leaf* components in Liquid.js: – **container components**: like a normal liquid component, a container component imports the liquid behaviour. Additionally it is possible to add into the containers any number of liquid components, they can be either other containers or leaves. Whenever a liquid primitive is invoked, the containers automatically broadcast the primitive invocation to all subordinated components. – **leaf components**: leaf components do not accept any subordinated liquid component. Whenever the application tries to create a liquid component inside of a leaf component, it automatically rejects the operation.

In the demo we will present how to create container components by importing the new *liquid container behaviour* into the Polymer behaviour list (Listing 1.1), moreover we will show what is the expected behaviour of the component in a live demonstration.

Listing 1.1: Liquid Container Paper-Input Component

```
<dom-module id="liquid-component-test">
  <template>
        ...
  </template>
  <script>
    Polymer({
      is: 'liquid-component-test',
      behaviors: [LiquidBehavior, LiquidContainerBehavior],
      properties: {
        ...
      },
    });
  </script>
</dom-module>
```

In this demo we will also present how it is possible to pair variables in Liquid.js. The pairing happens by invoking the *pairVariables* method passing two URLs into the method (*pairVariables(variableURL_1, variableURL_2)*). In fact all liquid variables in our framework are accessible by a unique URL (routing 1) which defines: – **device**: the device identifier which contains the component with the registered liquid variable; – **component**: the component identifier that registered the liquid variable; – **variable**: the name of the desired liquid variable.

$$/ : device / : component / : variable \tag{1}$$

Developers are allowed to use **wildcards** (*) whenever they write a variable URL. Routing 2 shows the routing that resolves as *all registered variables named text registered in all components contained in any devices.*

$$/ * / * / text \tag{2}$$

Moreover developers are allowed to write **[componentNames]** (surrounded by brackets) whenever they write a variable URL. Routing 3 shows the routing that resolves as *all registered variables named image inside the 'liquidImage' components in any device.*

$$/*/[liquidImage]/image \tag{3}$$

With this approach it is possible to pair liquid variables among any registered variable in the distributed application. Example 4 show a possible pair case in which the variable *image* registered by component *c1* contained in device *d1*, is paired with all other registered *image* variables in the system.

$$pairVariable('/d1/c1/image',\ '/*/*/image') \tag{4}$$

4 Conclusion and Future work

Liquid.js provides the default mechanisms to migrate flat and recursive applications between devices. In the future we will add a new level of abstraction to the variables URL routing, namely *:users*. In fact users own a set of devices, and by adding the users resource in our routing, it is possible to reference all devices owned by a single user. Moreover by introducing the concept of user in the system, we are also looking forward to implement *black* and *white* lists, which will increase the sharing security whenever users work with sensitive data.

Acknowledgments

This work is partially supported by the SNF and the Hasler Foundation with the Fundamentals of Parallel Programming for Platform-as-a-Service Clouds (SNF-200021_153560) and the Liquid Software Architecture (LiSA) grants.

References

1. Taivalsaari, A., Mikkonen, T., Systa, K.: 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, IEEE (2014) 338–343
2. Gallidabino, A., Pautasso, C., Ilvonen, V., Mikkonen, T., Systä, K., Voutilainen, J.P., Taivalsaari, A.: On the architecture of liquid software: technology alternatives and design space. In: accepted at WICSA'16. (2016)
3. Mikkonen, T., Systä, K., Pautasso, C.: Towards liquid web applications. In: Proc. of ICWE. Springer (2015) 134–143
4. Levin, M.: Designing Multi-device Experiences: An Ecosystem Approach to User Experiences Across Devices. O'Reilly (2014)
5. Gallidabino, A., Pautasso, C.: Deploying stateful web components on multiple devices with liquid.js for Polymer. In: accepted at CBSE'16