



BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS

ÚSTAV INFORMAČNÍCH SYSTÉMŮ

COMPLEMENTARY VIEW ADAPTATION WITH LIQUID.JS

PŘIZPŮSOBNÍ UŽIVATELSKÉHO ROZHRAŇÍ ZAŘÍZENÍM V LIQUID.JS

BACHELOR'S THESIS

BAKALÁŘSKÁ PRÁCE

AUTHOR

AUTOR PRÁCE

PETR KNETL

SUPERVISOR

VEDOUCÍ PRÁCE

Doc. Ing. ONDŘEJ RYŠAVÝ, Ph.D.

BRNO 2019

Bachelor's Thesis Specification



22208

Student: **Knetl Petr**
Programme: Information Technology
Title: **Complementary View Adaptation with Liquid.js**
Category: Web

Assignment:

1. Get familiar with Liquid applications, Liquid.js and Polymer. Learn Liquid.js API.
2. Design and implement the first version of the complementary view adaptation controller using the Liquid.js API.
3. Design a rule-based approach that allows developers decide how the user interface adapts to the set of available devices.
4. Extend the prototype with the proposed rule-based approach.
5. Develop a demo application to demonstrate the implemented prototype.
6. Evaluate the achieved results and discuss possible improvements.

Recommended literature:

- A. Gallidabino and C. Pautasso. Deploying stateful web components on multiple devices with liquid.js for Polymer. In Proc. of CBSE, pages 85-90. IEEE, 2016.
- A. Gallidabino and C. Pautasso. The Liquid User Experience API. In Proc. of the 27th International Conference on the World Wide Web (WWW), 2018.
- A. Gallidabino, C. Pautasso, T. Mikkonen, K. Systa, J.-P. Voutilainen, and A. Taivalsaari. Architecting liquid software. Journal of Web Engineering, 16(5&6):433-470, September 2017.
- E. Marcotte. Responsive Web Design. Editions Eyrolles, 2011.
- T. Mikkonen, K. Systa, and C. Pautasso. Towards liquid web applications. In Proc. of ICWE, pages 134-143. Springer, 2015.

Requirements for the first semester:

- Items 1 and 2.

Detailed formal requirements can be found at <http://www.fit.vutbr.cz/info/szz/>

Supervisor: **Ryšavý Ondřej, doc. Ing., Ph.D.**

Head of Department: Kolář Dušan, doc. Dr. Ing.

Beginning of work: November 1, 2018

Submission deadline: May 15, 2019

Approval date: October 31, 2018

Abstract

The thesis deals with design and implementation of the complementary view adaptation for the web framework Liquid.js. The introductory part summarises the idea of Liquid software and JavaScript framework Liquid.js. The goal of the thesis is the creation of the complementary view tool for Liquid.js which dynamically visualise the state of the user interface of liquid applications as an oriented graph. Moreover, it implements an additional interface that allows applying rule-based complementary view behaviours to liquid components. The requirements for complementary view and the proposed solution are described in the following practical part of the thesis. The text also talks about the implementation of the tool and describes the underlying programming paradigms and technologies used. The last section demonstrates the functionality of the final solution.

Abstrakt

Tato bakalářská práce se věnuje návrhu a implementaci Komplementárního zobrazení aplikací vytvořených pomocí webového frameworku Liquid.js. Úvodní část shrnuje a popisuje myšlenku tzv. "Liquid software" a JavaScriptového frameworku Liquid.js který na této myšlence staví. Cílem práce bylo navrhnout a implementovat Komplementární zobrazení Liquid.js aplikace, které zobrazuje aktuální stav aplikace a dovoluje jeho úpravu. Další požadovanou funkcionalitou je chování založené na pravidlech, které umožňuje předdefinovat chování aplikace za předem definovaných podmínek. Požadavky na řešení a návrh řešení je popsán v praktické části práce. Text dále popisuje použitá programovací paradigmatata a technologie použité při implementaci. Poslední kapitola demonstruje funkcionalitu finálního řešení.

Keywords

liquid software, liquid.js, complementary view, rule-based behaviour

Klíčová slova

liquid software, liquid.js, komplementární zobrazení, rule-based behaviour

Reference

KNETL, Petr. *Complementary View Adaptation with Liquid.js*. Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Doc. Ing. Ondřej Ryšavý, Ph.D.

Rozšířený abstrakt

S klesající cenou výpočetní techniky úměrně roste počet osobních výpočetních zařízení u každého z nás. V dnešní době průměrný uživatel internetu vlastní dvě chytrá výpočetní zařízení: osobní počítač a chytrý telefon. Čím více chytrých zařízení, tím složitější je jejich vzájemná komunikace a synchronizace. Jako řešení tohoto problému vznikla idea tzv. Tekutého software (z anglického Liquid software). Liquid software je zjednodušené software, který může být distribuován mezi více zařízení najednou a jeho jednotlivé komponenty mohou být přesouvány mezi těmito zařízeními. To uživateli umožňuje používat jednu aplikaci na více různorodých zařízeních bez nutnosti řešit kompatibilitu zařízení a synchronizaci dat. Liquid software je zatím pouze myšlenkou a zatím se nikomu nepodařilo ho vytvořit, přestože některé existující implementace se této myšlence více či méně blíží. Na myšlence Liquid software také staví webový framework Liquid.js. Ten umožňuje vývojářům vytvořit webovou aplikaci částečně odpovídající předpokladům Liquid software. To znamená, že taková aplikace je rozdělena do komponent, které jsou roz distribuovány mezi zařízení připojená k aplikaci. Připojená zařízení komunikují jak s webovým serverem, tak přímo mezi sebou skrze peer-to-peer spojení. Myšlenka Liquid software a webový framework Liquid.js jsou popsány v úvodní části práce.

Cílem této bakalářské práce bylo navrhnout a implementovat Komplementární zobrazení pro framework Liquid.js. Toto rozšíření dynamicky vizualizuje aktuální stav aplikace, tedy jednotlivá zařízení, na nich existující komponenty a synchronizace mezi nimi. Tato vizualizace zároveň umožňuje uživateli stav aplikace měnit skrze implementované uživatelské rozhraní. Další funkcionalitou Komplementárního zobrazení je možnost uživatele definovat pravidla, která se skládají z podmínky a akce (z anglického rule-based behaviour). Taková pravidla slouží jako předdefinované reakce na určitý stav aplikace. Podmínkou pravidla je určitý definovaný stav aplikace (např.: "K aplikaci je připojen jeden chytrý telefon."). Pokud se aplikace do takového stavu dostane, je provedena akce pravidla, která mění stav aplikace podle její definice (např.: "Na zařízení chytrý telefon přesuň komponentu klávesnice").

Prvním krokem byl teoretický návrh řešení. Bylo třeba identifikovat jakým způsobem bude grafická aplikace vyobrazena, jaké informace bude uživateli poskytovat a možností uživatele, kterými bude moci stav aplikace měnit. Dále bylo potřeba definovat termín pravidlo v rámci Komplementárního zobrazení. Proto bylo potřeba jasně vymežit strukturu pravidel, tedy všechny možné skladby a kombinace podmínek a akcí pravidla. Z toho důvodu byla vytvořena formální gramatika jasně definující jejich syntaxi. Také bylo zapotřebí navrhnout grafické rozhraní, skrze které může uživatel pravidla vytvářet a spravovat.

Dalším krokem byla již samotná implementace. Aplikace byla rozdělena do modulů pro jednodušší orientaci v kódu a navržena vzájemná komunikace těchto modulů a také komunikace se samotným API frameworku Liquid.js. Dále bylo potřeba navrhnout datové struktury aplikace a vybrat vhodné knihovny, aby implementace byla co nejjednodušší. Také musel být navržen způsob implementace celé logiky správy a vykonávání pravidel. Implementační detaily jsou popsány v předposlední kapitole práce.

V závěru práce jsou demonstrovány funkcionality finálního řešení. Tato část je rozšířena o snímky obrazovky zobrazující uživatelské rozhraní aplikace. U každého snímku jsou popsány grafické prvky které jsou na něm zachyceny a vysvětlena situace při které byl snímek pořízen.

Complementary View Adaptation with Liquid.js

Declaration

I declare that I have created this bachelor thesis independently under the direction of Doc. Ondřej Ryšavý from Faculty of Information Technology at Brno University of Technology and my advisors Andrea Gallidabino and Prof. Cesare Pautasso from Faculty of Informatics at Università della Svizzera italiana. I have listed all the literary sources and publications I have drawn from.

.....
Petr Knetl
July 29, 2019

Acknowledgements

I would first like to thank Doc. Ondřej Ryšavý for being my formal advisor of this thesis. Then I would like to thank my advisors Andrea Gallidabino and Prof. Cesare Pautasso from Faculty of Informatics at Università della Svizzera italiana for their valuable ideas, insights and advice.

Contents

1	Introduction	3
2	Liquid software	4
2.1	Requirements on liquid software	5
2.2	Use case scenarios	6
2.3	Architecture	7
2.4	Existing projects	7
3	Web framework Liquid.js	9
3.1	Liquid components	9
3.2	Components migration and communication	11
4	Design	14
4.1	Requirements	14
4.2	Graphical visualisation module	15
4.2.1	Layout	15
4.2.2	Controls	16
4.3	Rules logic module	17
4.3.1	Rule syntax	17
4.3.2	User interface	18
5	Implementation	20
5.1	Graph visualisation	21
5.1.1	Graph layout	21
5.1.2	Data storing and observers	21
5.1.3	User interactions	23
5.2	Rules handler	24
5.2.1	Rule creation	24
5.2.2	Rules storage	25
5.2.3	Rule deletion	26
5.2.4	Rule execution	26
6	Demonstration	28
7	Conclusion	31
7.1	Future work	31
	Bibliography	32

A Poster	35
B CD Contents	36
C Installation manual	37

Chapter 1

Introduction

Nowadays the average Web consumer owns at least two computing devices: usually a personal computer and a smartphone. Due to the decreasing prices of smart devices, is expected a further significant increase in the number of owned personal devices in any form (e.g. tablets or smartwatches). The more smart devices exist, the more complicated their mutual synchronisation and also the design of their mutual interactions become. As a solution to this problem, the idea of liquid software was created. Liquid software can be distributed between multiple heterogeneous devices and the components of the application can seamlessly flow between connected devices. Liquid software is described in [chapter 2](#).

The Web framework Liquid.js is built on this idea. Liquid.js allows developers to easily create liquid Web applications and liquid user experiences. The description of Liquid.js and its core functionalities of the framework can be found in [chapter 3](#).

The goal of this thesis is to create a tool that can display and manage the complementary view of any application built on top of the Liquid.js framework. The complementary view tool displays the dynamic multi-device deployment of the user interface of the liquid application and allows the user to change the deployment at run-time in real-time. The user can manage and change the state and deployment of the liquid components directly through the complementary view tool which directly exploits the Liquid.js API.

Finally, the tool also provides a rule-based approach for creating liquid user experiences that can automatically evolve and change the deployment of the application depending on the number and feature of the set of devices connected. The user of the complementary view has the opportunity of defining customisable rules that describe actions that should be performed on a liquid component whenever the selected condition is satisfied. The condition depends on the features provided by the set of connected devices. Whenever a condition is satisfied, then the rule action is triggered and the deployment of the liquid application changes. [chapter 4](#) proposes our solution and describes how is achieved the functionalities proposed above.

[Chapter 5](#) reports the implementation details of the solution. The chapter lists the technologies chosen at the design time, it shows the programming paradigms used, and it formally illustrates the entities built throughout this thesis.

The last chapter (see [chapter 6](#)) documents the functionality of the final solution by providing screen-shots which describe how to control the deployment of the graphical user interface of a liquid application.

Chapter 2

Liquid software

The liquid software metaphor refers to software that can be operated seamlessly across multiple devices. This metaphor was inspired by the ability of liquids to adapt its shape to the container holding it. More precisely, the Liquid Software can flow (e.g migrate) between devices without losing the state of the application ran by users [16].

The term liquid software was first used by Hartman, Manber, Peterson and Proebsting in a technical report back in 1996 [13]. This idea was taken over and shaped by the trio A. Taivalsaari, T. Mikkonen and K. Systs in 2014. The authors defined the Liquid Software Manifesto [22] which declares the key requirements which software must fulfil to be considered liquid. The authors formulated the manifesto as a reaction to the increasing number of smart devices in the world. They predicted that our digital world will turn into a world where people use dozens of devices at the same time during their daily lives. The devices can be of any form, such as: laptops, phones, tablets and „phablets“ of various sizes, game consoles, TVs, car displays, digital photo frames, home appliances, and so [12] – the only requirement is that these devices have to be able to connect and communicate with each other for instance through the internet.

Why is Liquid Software necessary? Already in the year 2008 the D. Dearman and J. Pierce comment in their study the difficulties with managing multiple computational devices [2]: „The collections of computing devices that people use to support their personal and business activities are growing. Instead of a single personal computer, users now incorporate multiple digital devices into their lives, including desktop computers, laptops, mobile phones, digital cameras and media players. Users also increasingly engage in activities that span devices, rather than just using different devices for different tasks. The resources for these activities, both information and applications, span multiple devices, and they may even be located elsewhere on the Internet. Employing multiple devices improves access to information and computation, but it requires managing information and activities across many devices, each with different limitations and affordances. Managing personal information and files is a significant issue for a single device; multiple devices exacerbate the issue.“ Simply the traditional software and operating systems are not designed to offer user experiences that take advantage of multiple devices [14].

Apart from the problem with difficult management and synchronisation of multiple devices by a single user, the trio F. Simon, Y. Landman and B. Sadogursky describes the current software market situation in their book about Liquid Software [20] as follows: „Continuous software improvement is not only about continuous development and deployment. It is an adjustment to how the marketplace operates. For the past several decades, the software has been sold as a commodity or a good. A customer would pay a price to

own a license for a piece of software or a software package. Revenue generated would be immediately transactional, with customers paying directly to acquire it. The marketplace is shifting, and it will continue to shift away from this model toward one in which software consumption is fluid and revenues are generated not as one-time payments, but as a constant stream, as users access a software service.“ The authors call this continuous shift from old markets models to liquid software as *liquid software revolution*.

According to the Liquid Software Manifesto[22], the exact definition of *liquid software* is: „an approach in which applications and data can flow from one device or screen to another seamlessly, allowing the users to roam freely from one device to another, no longer worrying about device management, not having their favourite applications or data, or having to remember complex steps. remember complex steps. In a true liquid software environment, device management chores such as backups, application installation, application upgrades, restarting the recently used applications, account migration, copying settings across devices, or other similar activities that burden the daily lives of users today should be things of the past.” [22]. The manifesto also summarises the key requirement for liquid software implementations:

2.1 Requirements on liquid software

1. The users shall be able to effortlessly roam parts of their application between all the computing devices that they have.
2. Roaming between multiple devices shall be as casual, fluid and hassle-free as possible. All the aspects related to the device maintenance and management shall be minimised or hidden from the end-user.
3. The user’s applications and data shall be synchronised transparently between all the computing devices that the user has.
4. Whenever applicable, roaming between multiple devices shall include the transportation and synchronisation of the full state of each application so that the users can seamlessly continue their previous activities on any other device.
5. Roaming between multiple devices shall not be limited to devices from a single vendor ecosystem only.
6. Finally, the user should remain in control of the liquidity of applications and data.

From these requirements, it is obvious that creating a liquid Web application that features all the described requirements is not technically possible yet. Moreover, the requirement number 5 describes a world where applications are vendor-independent and synchronisation may happen through any devices, which currently is not feasible. Nowadays many vendors (e.g. Apple) create their hardware and design their software ecosystems which do not support straight interaction with other systems. Although on the mobile smart devices the native apps still prevail, the focus of software companies is shifting toward web-based apps which is an important prerequisite for Liquid software [21].

2.2 Use case scenarios

The patterns of software usage are changing accordingly, as the users expect the possibility to access their data and application seamlessly on every device, possibly even using multiple devices at the same time [16]. The creators of the liquid software idea also described the basic use scenarios how a user can work with liquid software [10]:

1. **Sequential Use.** A single user runs an application on different devices at different times. The application adapts to the different devices capabilities while respecting the actual user needs in different usage contexts.
2. **Simultaneous Use.** A single user uses the services from several devices at the same time, i.e., the session is open and running on multiple devices at the same time. Different devices may show an adapted view of the same user interface, or the system may have a distributed user interface in which different devices play their distinct roles.
3. **Collaborative Use.** Several users run the same application on their devices. This collaboration can be either sequential or simultaneous.

The Sequential and Simultaneous uses were originally defined by Google in 2012 [11]. Figure 2.1 are presented example scenarios of sequential use case. In the left image, the user is transferring application from one tablet device to another. This scenario is also collaborative use case as two users are sharing the application. In the image on the right, the user is transferring the application state from tablet device to a car's navigation system.

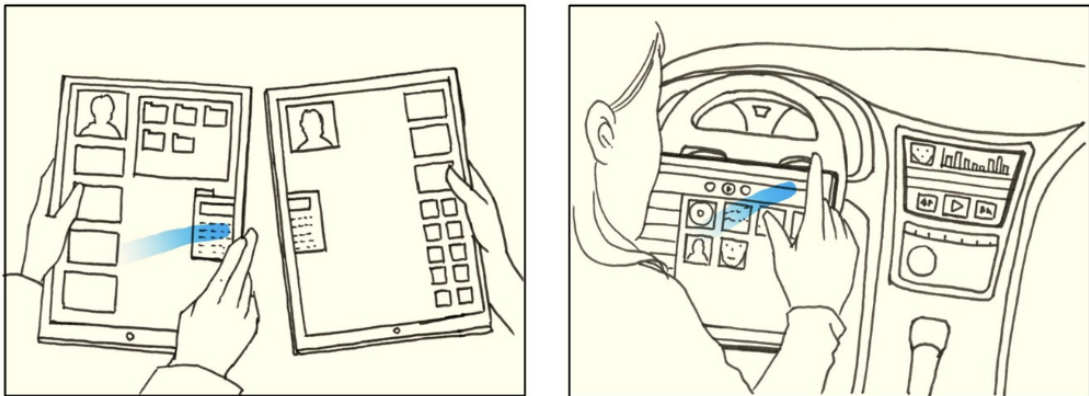


Figure 2.1: Examples of Liquid software Sequential Use Cases. Image was taken from [10].

In figure 2.2 are displayed examples of simultaneous use case. In both pictures are two various devices running the same application. These devices are synchronised, so any action executed on one device will be also displayed on the second device. The user is using the smartphone as remote control of application running on the tablet device, similar to TV spectator can use a remote control to change volume or TV channel.

To make Liquid software able to run on a variety of smart devices, the visual design of the application has to be adaptive to screen of any size. This problem can be partially solved with implementing Liquid Software supporting *responsive web design*[17].

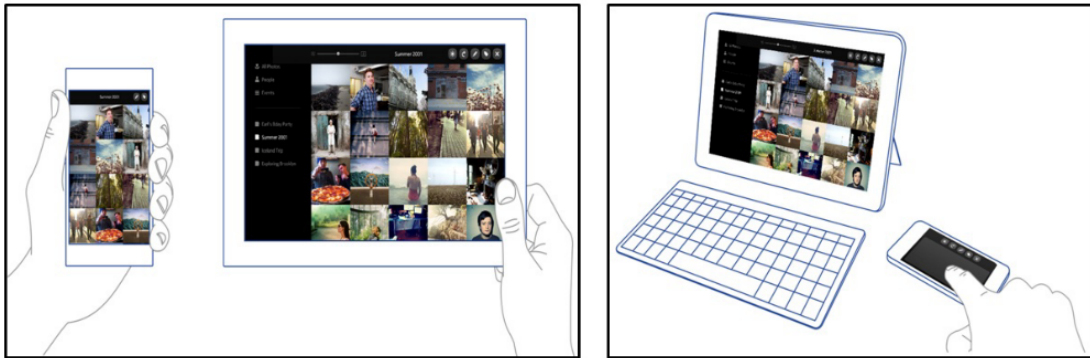


Figure 2.2: Examples of Liquid software Simultaneous Use Cases. Image was taken from [10].

2.3 Architecture

There are two possible types of topology for Liquid Software. One of them is *centralised topology* [23] with a single host (server) that maintains the master copy of the application. This centralised host is usually available in the cloud, taking advantage of the high availability and virtually unlimited capacity of data centres, potentially at the expense of the privacy of the data that is no longer confined only to user-controlled devices. Liquid Software thus flows up and down from the cloud onto various user devices (clients) that are thereby implicitly backed up and synchronised as long as a connection to the cloud is available. Alternative solution is *decentralised topology* [24]. With the use of this architecture, the data and state of the application are shared directly between connected devices in a peer-to-peer fashion. This basic topology decision – centralised versus decentralised design – can be regarded as a fundamental dimension in the context of Liquid Software. Granted, with a central server, it is easier to manage software as well as data content. However, the decentralised alternative can offer significant benefits as well, since only local connectivity is needed for migrating state from one device to another, and the user’s data can be kept outside the reach of major cloud providers [10]. Both of the proposed typologies are visualised in figure 2.3.

2.4 Existing projects

In the past, several “semi-liquid,, projects were created, fulfilling some requirement described above. For example, Apple’s iCloud, the cloud storage solution for iOS devices. This service can store files, settings, contacts, bookmarks and even installed apps in the cloud and share those data easily between different devices. This project is partly liquid because it fulfils some of the requirements on the liquid software (2.1), but not all of them.

Another project was Sun’s (now Oracle’s) Sun Ray platform. Users using this software could easily transport their entire network computing session between physical Sun Ray terminals (clients). The Sun Ray terminals are physical stations which do not perform any computations. Apart from the physical terminal, there is also a client’s instance running on the server side where are all the computations done. The physical client sends user actions to the server. Then the server streams back the compressed live image of the screen which

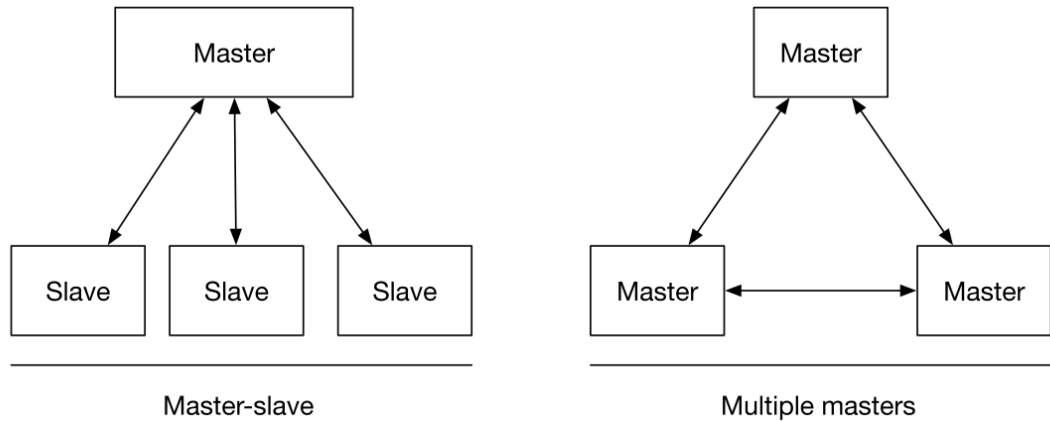


Figure 2.3: Illustration of Server-client (on the left) and peer-to-peer (on the right) topology.

is displayed on the client's station. The authentication of the client instance is based on inserting JavaCard into the terminal. Because no raw data are shared via the network, Sun Ray terminals are very secure, but their performance is highly dependent on the network connection [19]. Again, this project is just partly liquid. None of these mentioned systems fulfils all requirements described in section 2.1. The vision of liquid software in the future may not be limited to only specific liquid applications, but a whole liquid operating system. It would be possible to fragment such system into many components that can be distributed across multiple devices according to the user needs. This is not yet technically possible. The closest implementation of liquid software that can fulfil all the requirements are liquid web applications, which rely on internet browsers that can run on any Web-enabled device.

There also exists frameworks that allow creating liquid web applications. One of them is PolyChrome [1] which makes the whole web application liquid. The different approach takes a framework called Liquid.js [6] which targets component-based applications, so only the specified components behave as liquid [8]. The framework Liquid.js is the topic of the next chapter.

Chapter 3

Web framework Liquid.js

As described in the previous chapter, the increasing complexity of synchronisation multiple smart devices has to be solved. Nowadays the smart devices can connect to the internet and have enough computational power to execute complex tasks which in the past were impossible, but the outdated system architectures hold the devices back. “All these devices are connected to the Web, and their users are provided with computation that is constantly available, capable of delivering meaningful value even in few moments, without requiring active attention from the users part. The architecture of current Web applications is not living up to these expectations. Content is increasingly made available on the Web and users have many ways to access the content published on the Web, but they are exposed to extra complexity caused by a large number of Web-enabled devices at their disposal. Additional complexity comes from the fact that user content is spread to several devices and Internet services. Managing all these as separate entities is currently a tedious task, while at the same time users expect casual experiences. Since all these devices are already connected to the Web, orchestrating their actions to simplify users’ lives would be a natural extension of the Web platform.”, Describes T. Mikkonen, K. Systa and C. Pautasso [18].

„Traditional (non-liquid) applications are usually designed with a server-centred architecture, whereby all persistent data, dynamic states and logic of the application are stored in the central server of the application. The client mostly works as a mediator between the user and web server, displaying pre-computed views sent from the server.“ [15]. The framework Liquid.js implements a different approach. It builds on the idea of liquid software as described in the requirements listed in [chapter 2](#). The framework supports the development of Web applications for sequential and simultaneous screening scenarios across multiple heterogeneous Web-enabled devices (these devices has to be able to run a modern Web browser). Usually the multiple devices are owned by the same user [6], however, in collaborative use case scenario, multiple devices can be owned and controlled by multiple users. Liquid software allows the users to run the application on the whole set of devices and enhances the liquid user experience in a convenient collaborative way [9].

3.1 Liquid components

Liquid applications built with Liquid.js are structured into *Liquid components*, which allows to seamlessly migrate parts of an application across the set of connected devices [7]. Liquid components can be deployed and distributed across any device connected, and even when they roam, they can still share their data with other components and synchronise their



Figure 3.1: Liquid.js logo. Image taken from [4].

state with other components. The component behaviour is application-specific and both the HTML/CSS and logic is implemented by the developers that choose to use the Liquid.js framework API to manage the deployment of the liquid application, as well as the replication and synchronisation of its state [8].

The components exist on connected devices and inside them exists *liquid properties* defined by name, value and permission. Figure 3.2 visualise the hierarchy of components and properties. The properties permission can be set either to *publish* or *subscribe*. When the property permission is set to subscribe, the component is authorised to fetch the data from other paired components. In the case, the option is set to publish, then the component is permitted to update the properties of subscribing paired components, but it is unable to read the property values of other components. Permission settings also work as a privacy feature, as it keeps certain data private and inaccessible from malicious or defected components [7]. All properties defined inside a liquid component, together they define the *state of a component*.

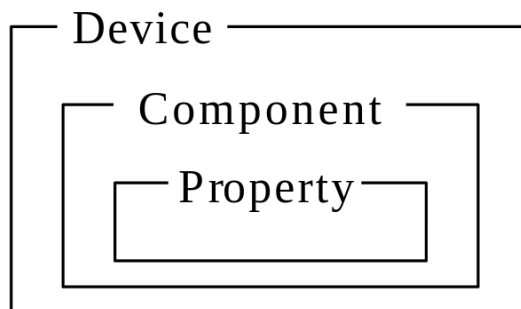


Figure 3.2: Hierarchy of devices, components and properties according to Liquid.js architecture.

The value held by liquid properties on a device can be stored in different places according to the implementation. The options are *Browser Memory* (JavaScript Heap), Local Storage, Session Storage and Server-side storage. *Browser memory* is used for volatile values which need to be stored until at least one instance of all the components holding it exists. The *Local storage* is a good solution for property values that need to be persisted on a device. If

the local storage is not supported, the Server-side Storage is used instead. *Session Storage* is similar to local storage with the only difference that the session storage content is discarded when the browser session is closed. *Server-side Storage* should be used for saving global variables which can be potentially needed by any existing component [6] instantiated in the application.

The persistence of properties can be set to *persistent*, *session* or *volatile*. If the *persistent* option is set, the value of the property is preserved even if all the components using this value is closed. The *session* option preserves the value until at least one web browser which uses or used the value is opened. The *volatile* memory is deleted when there is no component using it.

Every liquid component encapsulates Polymer component visualise the layers , which provides the transparent automatic state synchronisation behaviour between multiple components instances [6]. The figure 3.3 visualise the layers of a smart device running Liquid.js. Liquid.js is compatible with any Polymer element, which allows the developer to use pre-defined Polymer components found at *Catalogue of Web Components*¹. That provides easy reusability of already existing components.

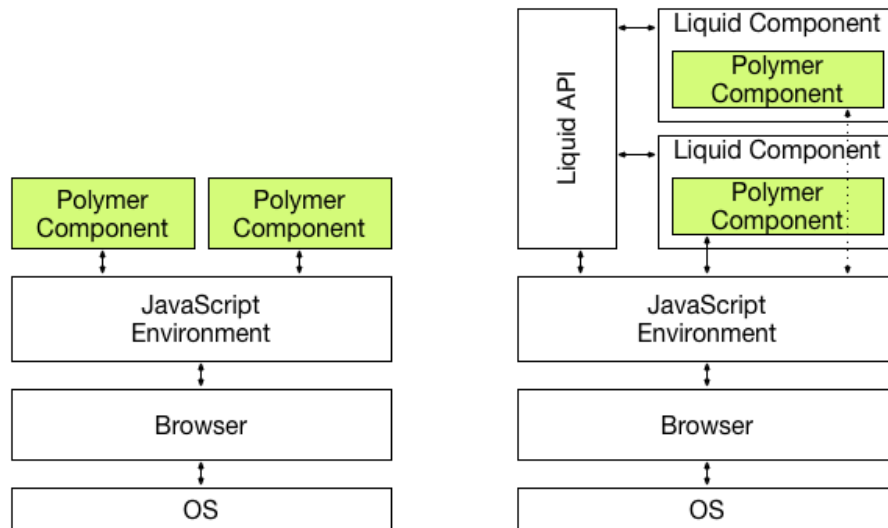


Figure 3.3: The layers of system running web application with use of Polymer components (on the left) and Liquid.js web application (on the right). Liquid components encapsulates the polymer components. Image taken from [4].

Every device, liquid component and the liquid property is identified with a unique id which can be used as a referral to a specific object of the application (*URI*). The id and the URI are assigned at the moment of creation [5].

3.2 Components migration and communication

The framework supports all the use-case scenarios described in section 2.2: *sequential screening*, *simultaneous screening* and *collaborative* scenarios [9]. *Sequential screening* is designed single users who own multiple devices and decides to use an application with a

¹Catalogue of Web Components <https://www.webcomponents.org/>

single device at the time. This scenario allows the user to migrate the application from one device to another and resume what he is doing on a new device. *Simultaneous screening* describes a scenario where a single user, who owns multiple devices, decides to use all his devices at the same time and expects the application components to be distributed across them. *Collaborative* scenarios describe the interaction of multiple users who own multiple devices and use all of them at the same time in a collaborative environment.

Liquid.js allows choosing two different communication channels between devices. If the devices support WebRTC data channels², the communication is created using the WebRTC standard protocol. The components can then exchange peer-to-peer messages directly among them. This kind of communication is very advantageous because the server does not rely on messages to the client and it is relieved of some traffic. Whenever WebRTC communication is not available, the communication is directed through the server. Independently from the communication channel, the liquid server is always used for device discovery and pairing devices when new WebRTC communication must be established [6].

The user of a liquid application has access to three liquid primitives that can be used to move components to another device. The first primitive is called *migration* (figure 3.4). This option deletes the component from the original device and creates a new instance on the device specified by the user. The second primitive is called *fork* (figure 3.5). Whenever the fork operation is called, the original component stays on the original device, but the Liquid.js creates a copy of the component on the target device. The last primitive is called *clone* (figure 3.6). It is similar to fork primitive, but besides, the original and new components are paired and stay synchronised over time. All changes produced in the state of a component are automatically propagated to the properties of all paired components [8].

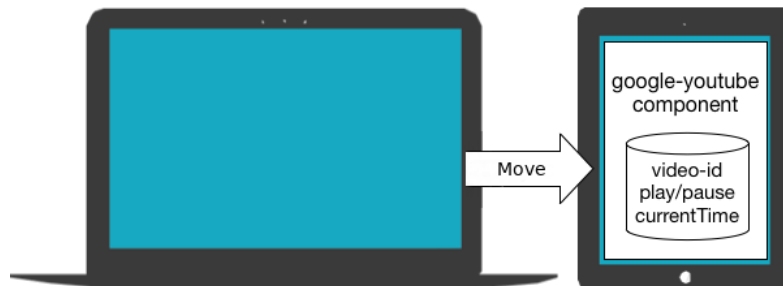


Figure 3.4: Demonstration of migration primitive. The component is moved from laptop device to the tablet device. Image taken from [4].

Apart from the possibility to move and synchronise whole components, the Liquid.js also implements primitives for synchronisation of individual component properties. It is possible to use *pair* primitive, to connect and synchronise any two properties. The inverse primitive *unpair* cancel existing synchronisations. With all these previously mentioned primitives, the developer has full control of deployment of the application and its states.

²WebRTC <https://webrtc.org/>

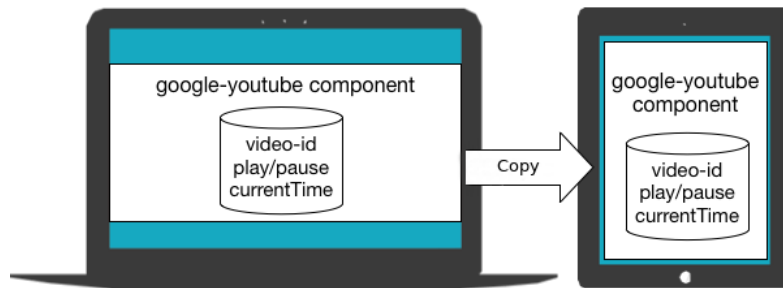


Figure 3.5: Demonstration of form primitive. The component is copied from laptop device to the tablet device. Image taken from [4].

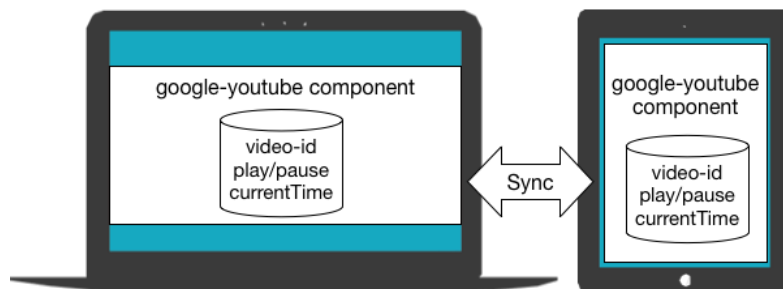


Figure 3.6: Demonstration of form primitive. The component is copied from laptop device to the tablet device. furthermore the newly created component is also synchronised with the original one. Image taken from [4].

Chapter 4

Design

This chapter describes the design of our proposed solution of the complementary view implementation. First of all, it is necessary to clearly describe the requirements of the final solution. Then from the requirements, it is possible to create a proposal of the solution and design the structure of the application. The implementation of the tool can be started only after a reasonable and feasible application is designed.

4.1 Requirements

First of all, we need to find out what is the purpose of the desired application. So it is necessary to find out the requirements of the application functionalities. The main feature of the application is defined by the needs of the developers to understand how to build the liquid user experience of their applications, and the needs of the administrators and the users that need to manage and have control over the deployment of the UI of liquid applications. The goal of the complementary view is to give them a tool which allows seeing in realtime the deployment of the whole application in a single visualisation.

Through this visualisation, they can actively interact and change the application state with the use of the underlying Liquid.js API. The user needs to get an overview of the connected devices, a description of the kind of connections between them, and the information about all liquid components instantiated. Every component may contain multiple properties, which could be connected and synchronised between each other. This information must be visible as well. Moreover, the visualisation also provides to the user basic information about the connected devices, such as the browser version, the device OS, and the type of the connected device (e.g. laptop, or smartphone). The user also has to be able to change the state of the deployment directly through the tool. The user can choose to migrate, clone, fork or delete components between devices, and he can choose to pair or unpair liquid properties.

Last but not least, the complementary view implements a rule-based approach to allow user plan actions changing the deployment of the application according to the current state of the application. That means that the user can predefine the behaviour of the application reacting to changes in the application state. The user has to be able to create custom rules defining a condition and an action. The condition of the rule describes the deployment configuration of the application. If the application evolves to such a deployment state, then the action is triggered. The possible rule actions are: migrate, fork, clone and delete an instantiated liquid component and pair/unpair two liquid properties. The user can also

activate, deactivate, delete the rules, or change the priority of a rule relative to other defined rules.

All these user actions should be executed transparently without any users knowledge of the actual Liquid.js API. That means that the complementary view has to hide the Liquid API under the graphical user interface of the complementary view tool.

After the definition of the functional requirements, it is necessary to derive the appropriate logic of the application. The view has to implement two main features: the *Graphical visualisation module* is a graphical user interface that represents the current state of the deployment of the application. The user can also directly modify the deployment of the application. The graphical visualisation module can directly communicate with the Liquid.js framework via its API. The second feature is to create the logic behind the rule-based approach. This feature will be hereinafter referred to as *Rules Logic module*. The term „rule“ has to be formally defined, after that, the procedures of the rule creation and rule execution can be designed.

4.2 Graphical visualisation module

The Graphical visualisation has to represent all devices connected within the application, the instantiated components and the liquid properties. Whenever the state of the application changes, the visualisation has to dynamically reflect these changes as well.

I decided to visualise the deployment state as an oriented graph. The devices, components and properties will be displayed as *graph nodes*. Every device node will contain all the *component nodes* present on the device. Inside the component, nodes will be displayed another level of sub-nodes representing the liquid *properties nodes*. These sub-nodes will represent every liquid property of the component. Because properties can be paired and synchronised, then it is necessary to visualise such connections as well. The pairing of properties will be represented by *oriented edges* between the source property node and target property node of such a pair.

4.2.1 Layout

To create graph visualisation, a suitable layout of the graph needs to be designed. I decided to design the *device nodes* as a rectangle divided into two parts. Both of these parts will have a rectangle shape and same width as the device node. The first part hereinafter referred to as the *Informational part* will hold important information about the device such as the type of the device (e.g. desktop, phone, tablet, etc.), the browser version, the operating system and the device performance score. The second part hereinafter referred to as the *Components wrapper* will be a rectangle aligned with the Informational part and positioned right underneath. Within the wrapper will be displayed all the existing *component nodes*. Each component will also be displayed using a rectangle shape, it will have a predefined width and it will be ordered and positioned underneath each other. Inside the component nodes, an additional layer will be displayed: the nodes representing the liquid properties. *Property nodes* will be represented with a rectangle shape and listed underneath each other. The pairing of the two properties will be displayed as an arrow. If the source and the target properties are deployed on different device nodes, the line will be straight. If the device node is the same, then the line will be shaped as a Bézier curve. To make the visualisation more clear, I have decided to create functionality which allows users to *highlight* specific device directly from the graphical visualisation tool. When the user executes the highlight

action, then the node representing the highlighted device will change colour. The user can choose how the highlight will be triggered on the highlighted device.

The visualisation layout as described in this chapter is shown in [Figure 4.1](#). In this example, there are two devices connected to the application. One of them is a desktop running Mac OS, the second one is an Android smartphone. On the Desktop device, two components are instantiated: a chat and a picture component. On the Android device, only a picture component is instantiated. There are two pairings present. On the desktop device, there is a connection between the history property of the chat component and text property of the picture component. The other connection is across devices. The image property of the picture component from the desktop device is paired with the Image property of the picture component existing on the phone device.

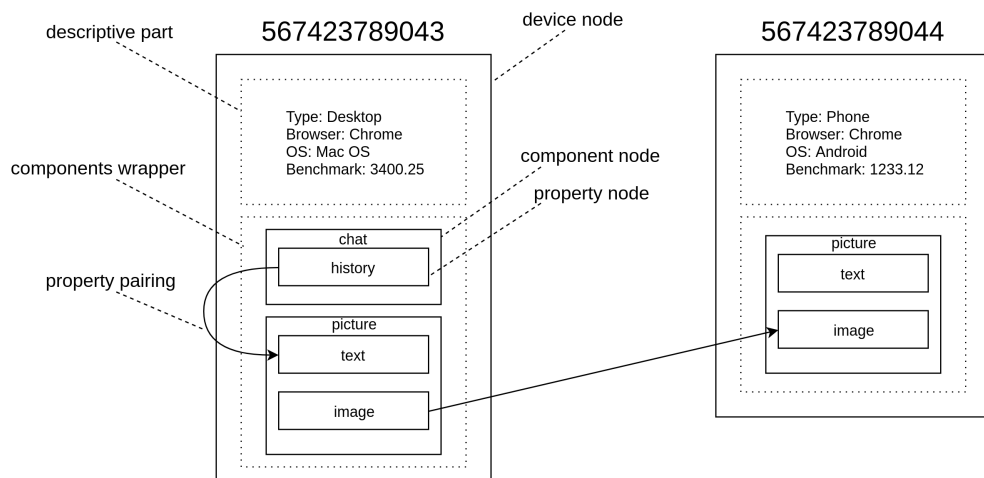


Figure 4.1: Draft of complementary view graphical layout.

4.2.2 Controls

Through the visualisation, the user should be able to directly change the state of the application. The user should also be in control of the position of the components and the pairings between properties. The implemented operations that can be applied to components will be: migrate, fork clone, and delete. The *migrate*, *fork* and *clone* operations can be executed by dragging the desired component node on top of any device node and then by selecting one of these operations from the context menu. If the user wants to *delete* a component, it will be possible to achieve that by right-clicking on the component node and selecting delete operation from the context menu. From the same context menu it is also possible to *enable* or *disable the highlight* of a device.

The user needs to be in control of the pairing of the liquid properties. To create a new pairing between properties, he will need to left-click on top of the source property node and drag it to the target property node. The existing pairing can be deleted by right-clicking on edge representing the connection and by selecting the delete operation from the context menu.

Apart from the previously described controls, the user should also have the possibility to navigate across the graph. For example, if the graph is too big, and it overflows the display borders, then it would not be possible to intuitively use the graphical visualisation. Because of this reason, the user should be able to *pan* and *zoom* the graph. To zoom, the user can use the mouse wheel or pinch-zoom gesture if is using a touchscreen. Panning will be possible through drag and drop in the area where no nodes are present. The zooming and panning will also be implemented in the form of GUI controls that will be found on the corner of the graphical visualisation.

4.3 Rules logic module

The goal of this module is to allow the user to define rule-based behaviour which allows dynamically change the deployment state of the application. That means, that user can manipulate the deployment state not just in the present moment, but he also can define actions which will be executed in the future if a specified condition is met.

4.3.1 Rule syntax

To be able to describe and create the rule-based behaviour we need to define the term *rule* in the context of the complementary view first. The rule is an executable statement consisting of a source entity, an action, a target entity and a condition. I have decided to divide rules into two categories. One category describes rules that can be applied on components and the second category can be applied on properties. Both of these categories have a defined structure. To formally describe this structure, I created a Backus–Naur form grammar which describes the possible syntax of a rule. The grammar is shown in Figure 4.2.

The structure of the condition is the same for both the component and the property rules. The condition describes the required state of the deployment that is needed to trigger the specified action. More precisely it describes the number and types of connected devices. The rule condition is composed of three parameters: the device type, the count and the operator. The *device type* parameter describes which type of devices is monitored by the condition. This parameter can hold four types of value: *Desktop*, *Phone*, *Tablet* or general *Device*. The *count* parameter has to be a positive integer and express the threshold count of devices under which the condition is evaluated as fulfilled. The last but not least part of the condition is the operator. The *operator* can hold the value of any logical comparison operator, thus: *equal to*, *not equal to*, *greater than*, *lower than*, *greater or equal to* or *lower or equal to*. The operator describes the relationship between the state of the application and the count parameter of the condition. The example of valid rule condition could be: „Device“ 4 „ge“. This condition expresses the state of application where the number of connected devices is greater or equal to four.

The condition is the only part which is common for both the component and the property rules. Component rules describe two more parts: the source component and the component action. The *source component* defines the instantiated liquid component by its unique Liquid.js id which will be affected by the defined action. The *component action* has two parameters. First parameter is the *action keyword* which maps to Liquid.js primitives. The possible values of action keyword are: *migrate*, *clone*, *fork* and *delete*. In case of *migrate*, *clone* or *fork* the second parameter targets a device. The *target device* holds value of a unique Liquid.js device id, or of a type of device. In the case of type of devices the possible

```

⟨rule⟩ ::= ⟨componentId⟩ ⟨componentAction⟩ ⟨condition⟩
        | ⟨propertyId⟩ ⟨propertyAction⟩ ⟨condition⟩

⟨componentAction⟩ ::= clone ⟨targetDevice⟩
                  | fork ⟨targetDevice⟩
                  | migrate ⟨targetDevice⟩
                  | delete

⟨targetDevice⟩ ::= ⟨deviceId⟩ | ⟨deviceType⟩

⟨deviceType⟩ ::= Desktop | Phone | Tablet

⟨propertyAction⟩ ::= pair ⟨targetProperty⟩ | unpair ⟨targetProperty⟩

⟨targetProperty⟩ ::= ⟨propertyURI⟩

⟨condition⟩ ::= ⟨operator⟩ ⟨unsignedInteger⟩ ⟨deviceType⟩

⟨operator⟩ ::= eq | ne | gt | lt | ge | le

⟨unsignedInteger⟩ ::= ⟨digit⟩ | ⟨unsignedInteger⟩ ⟨digit⟩

⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

Figure 4.2: Grammar describing rule statement syntax.

values are: *Desktop*, *Phone*, *Tablet*. If the selected action is delete, then then target device parameter has to be empty.

The property rules are similar to component rules. Instead of the source component, they define source properties. The source property contains the unique Liquid.js id of a property. The action of properties rules always consists of two parameters: action keyword and target property. Action keyword has two possible values, *pair* and *unpair*. The target value defines the unique Liquid.js id of a property. An important constraint is that the source property and target property has to be different properties, otherwise the rule is not valid.

4.3.2 User interface

It is necessary to specify how the user can create new rules and manage the existing one. To create new rules, I designed a dynamic input form. The menu consists of multiple drop-down menu inputs. Gradually, as the user fills the inputs, the options in the empty inputs dynamically change. This behaviour prevents the user from defining invalid rules.

First of all, the user has to define the source entity of the rule. For this purpose, the first input lists all the instantiated components and properties as possible source options. If the user selects a component, then the second input reacts and changes its options to match component actions, thus it shows the migrate, fork, clone, and delete options. After the selection of the component action, the third input allows the user to specify the target device of the action. In the case of a delete action, the third input is disabled. The last

three inputs together create the rule condition. In the fourth input, the user specifies the operator, in the fifth, he must insert the right number of devices and in the sixth the type of device(s). For properties, the values of the second input listed are pair and unpair. The third input prompts to user target properties. The last three inputs are the same as for component rule definition.

For managing existing rules, I will design a simple list. In this list, every rule will be represented by a string describing the rule. On the right side next to the text, there will be trash bin icon allowing to delete the rule. User will be able to change the priority of the rules by changing their order in the list. This will be possible to achieve by drag and drop of the rule representation in the list.

Chapter 5

Implementation

The first step is to define the modules of the application and their mutual interactions. I have decided to divide my implementation of the complementary view tool into two main modules. The first one is called *Graph Visualisation*, which is responsible for rendering the graphical overview of the application deployment state and handles the user actions which can alter the deployment. The second module is called *Rules Handler* and it extends the functionalities delivered by the first module by adding a rule-based behaviour for automatically manage the dynamic deployment of the application. As explained in [section 4.3](#) it is responsible for defining, storing and executing the logic behind the rules system. The component view of the implemented system is shown in [Figure 5.1](#).

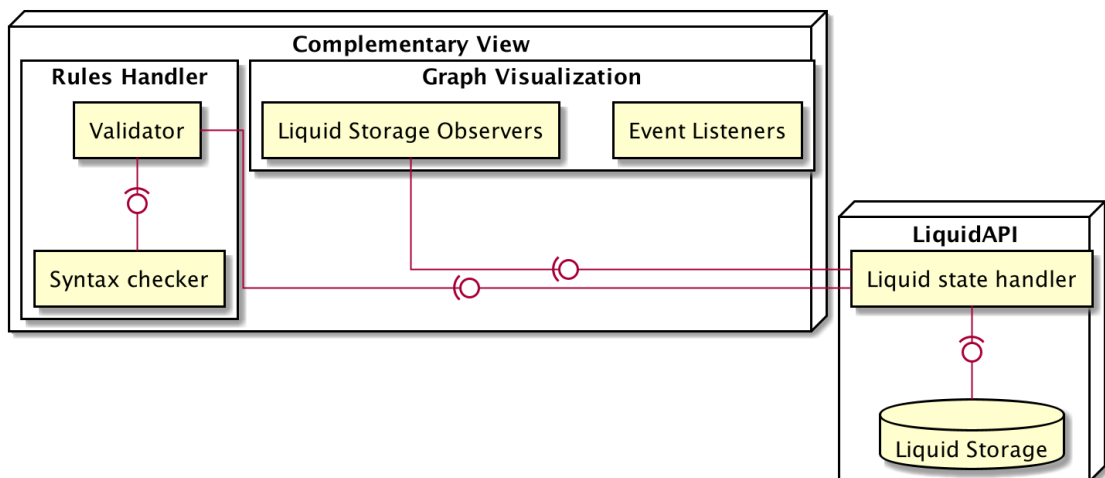


Figure 5.1: Component diagram of the Complementary view tool.

Since the complementary view tool is built on the web, it is entirely implemented with the use of JavaScript, HTML and CSS languages. The core HTML template of the application is integrated into a Polymer component extended with the Liquid.js behaviour. This allows to treat the Complementary view tool like any other liquid component, thus the complementary view itself can be migrated, cloned, forked or deleted through the Liquid.js API. The overall visual appearance of the complementary view is defined by CSS style sheets with the use of the *Bootstrap*¹ framework, which simplifies the creation of the

¹Bootstrap – <https://getbootstrap.com/>

style sheets and offers predefined styles. In order to simplify HTML manipulation with, I also used the JavaScript library *jQuery*². The library provides an easy to use API which simplifies many of the JavaScript features.

5.1 Graph visualisation

The core functionality of this module is displaying the graphical representation of the application deployment state. The proposed solution suggests the visualisation in the form of an oriented graph. Since implementing the whole logic of the interactive oriented graph from scratch would be complicated and time-consuming, it is necessary to find an existing library which allows creating and dynamically modify graphs. For this purpose, I chose *Cytoscape.js*³. In the official documentation, the Cytoscape.js is described as fully featured open-source graph theory library. Cytoscape.js can be used for graph analysis and visualisation. It also allows the user to easily display and interactively manipulate graphs through its API [3]. Another useful feature of Cytoscape.js is that it is fully extendable. That means it allows to create and share custom extensions among the community. I used some of those extensions as I will describe later.

5.1.1 Graph layout

When a new device is connected and its node created, we need to decide where will be the node positioned. For this purpose is implemented an automatic templating function which decides the position of the newly created node according to the position of the already existing device nodes. If there is no existing device node, the new node is positioned in the middle of the graph. If there is at least one existing device node, the function reads the position of the right-most one and sets the position of the new device node to be rendered on the right side of the right-most one. For the positioning child nodes inside their parent nodes, I implemented the layout function which is positioning the component nodes inside the device nodes and property nodes inside the nodes of the component according to the proposed solution. This function takes as a parameter a unique id of device node and sorts all its descendants inside its body. The layout function is triggered whenever the observer detects the addition or deletion of the component of the device. I configured Cytoscape.js visualisation to be unable to change descendant position inside of parent node's body. That means after the execution of layout function, the user is unable to shuffle the components nodes position in the device nodes and properties inside the component nodes. The final layout after the execution of the function corresponds with the proposed layout described in [subsection 4.2.1](#).

5.1.2 Data storing and observers

To be able to visualise the state of the application, we need to access all the meta-data describing the distribution of components between devices. Every Liquid.js device contains a component called Liquid storage. This module is implemented with the use of *Yjs*⁴ JavaScript framework. Yjs is a framework for offline-first p2p shared editing on structured data. It creates the peer-to-peer connections between all the individual storage located on

²jQuery – <https://jquery.com/>

³Cytoscape.js - Graph theory library for visualisation and analysis – <http://js.cytoscape.org/>

⁴Yjs <http://y-js.org/>

every existing Liquid.js component. I decided to use this synchronised storage for storing the meta-data describing the current application deployment state. From this meta-data, the complementary view component can construct the graph representation of the application. The meta-data is stored in a JavaScript object like structure. Yjs calls these synchronised objects a *Y.Map*.

Data in the map is stored in the following structure. For every connected device is created a property. This property contains another Y.Map. Inside this inner map are three properties. One called *highlight* containing the Boolean value expressing if the device is currently highlighted, the second property *position* is holding object value containing the coordinates of the device node representation in the graph visualisation and third property is called *components* with value of the another map holding information about every component present on this specific device. Every component is represented again as a map containing the last level map for every liquid property inside of the component. In the liquid property, the map is stored information about pairings with other properties. The structure of liquid storage is shown in listing 5.1.

```
{
  deviceId: {
    highlight: boolean,
    position : {x: integer. y: integer}
    components:{
      componentId: {
        propertyId:{
          incoming: [propertyId, propertyId, ...],
          outgoing: [ propertyId, propertyId, ...]
        },
        ...
      }
    },
    ...
  },
  ...
},
...
}
```

Listing 5.1: Liquid state data structure

I had to modify the Liquid.js framework to automatically update this map structure whenever any important action for graph visualisation is executed. Now whenever the new device is connected to the application, the Liquid.js creates the Y.Map describing this device. Then whenever a new component is created, deleted, paired, migrated or cloned the change is reflected in the Liquid storage map as well. Also, the pairing and unpairing of properties are reflected in the map.

Yjs also provides functionality called Observers. An observer is a JavaScript object monitoring the events influencing the Y.Map. If are the data of the map modified in any way, the observer catches this event and executes an user-defined callback function with the data describing the occurred event as a parameter. Inside the graph visualisation module, i have defined such observers which are reacting on my events triggered inside the Liquid.js.

These observers are a crucial part of graph visualisation module. They are evaluating the incoming events and calling the Cytoscape.js API to edit the graph. Whenever a new device is created, the observer reacts with creating a new node representing the device in

the graph. Similar action is executed when a new component is created or moved from one device to another. If the component contains any liquid properties, then the nodes for the properties are created as well. When the event of pairing is caught, then the Cytoscape.js edge representing such a connection is created. On the other hand, the edge is deleted if Liquid.js triggers an unpair event.

5.1.3 User interactions

Another important part of Graphical visualisation module are event listeners handling users interaction with the graph. When the user right-clicks on the node, the context menu with options according to the type of the node appears. For implementing this functionality I used Cytoscape.js extension called Cytoscape-context-menus. This extension provides a simple API for defining the context of different options according to the type of node or edge selected. It takes the list of objects. Every object in this list represents one context menu option. This object contains the option text label, Cytoscape.js text selector defining for which nodes the option will be displayed and a callback function triggered when the option is selected. This function is executed when the option is selected. I use these callback function to calling Liquid.js API. That means, that this context menu UI allows to the user directly communicate with Liquid.js and modify the state of the app. For example delete component with selecting delete option from the context menu displayed by right click on the component node.

For device nodes, the context menu options are *highlight* an additional option for every possible component type in running application apart from complementary view. If the highlight option is selected, then the corresponding device property *highlight* inside the Liquid storage component is updated and set to true. The colour of the displayed node is changed to green and the executed context menu option is changed to *unhighlight*, which allows reverting the highlight action. When the user selects the *add component* action, the Liquid API is called and the desired component is created inside the device where the action was requested.

For component nodes, there are only two context menu options. The first one is *delete*, which calls the Liquid.API and deletes the selected component. The second option is used to create and assign a new rule to the component. Whenever the user clicks this option, a new menu opens and the form used for the new rule definition, filled with the predefined value of the source component which triggered the event, is displayed. This form is described in section [4.3.2](#)

Another implemented group of event listeners are mouse events, specifically drag-and-drop events. There are three uses of such an event: First of all, it is used for *panning the graph*. This functionality is built inside the Cytoscape.js. Then I implemented *drag-and-drop action for properties pairing*. If the user drags mouse cursor for one property to another one and then drops it, the pairing is created. Similar functionality works for *migrating, forking and cloning the component*. After drag and drop component to the different device node, the context menu appears. This context menu contains three options: *migrate*, *fork*, *clone*. After selecting one of these options, the corresponding Liquid.js API method is called. The last use of these events is *re-positioning of the device node* in the graph visualisation as a drag-and-drop action.

5.2 Rules handler

This module serves all the functionalities connected to the rule-based behaviour provided by the complementary view tool. That means that this component handles the creation, verification, storing, execution, and deletion of the rules. To achieve this functionality the module needs to directly communicate with the Graph Visualisation module and Liquid.js storage.

5.2.1 Rule creation

As described in chapter 4.3.2, I created an HTML form that allows the user to create new rules. The dynamic behaviour, that changes in real-time the values of the form reacting on the user inputs, was implemented in JavaScript. When the user opens any of the form inputs, a dedicated event listener function is triggered. This is achieved with the use of the standard JavaScript event listeners API. For the first input, the function needs to load a list of all the instantiated components and properties in the graph. To do this, the function access the Cytoscape.js API through the Graphical visualisation module and reads the stored information about the existing components and property nodes. Then for every found node, the function creates an option inside the input field. After selecting an option the second field of the form loads action options, different for components and different for properties as described in chapter 4.3.2. After selecting the action, the third input loads possible target properties or devices in case of the components. To do this the loading function accesses the Cytoscape.js API a second time and reads the information about the existing nodes. Then the last three inputs are activated. Their options are always the same because the value of the condition is independent of previously selected values.

```
{
  sourceEntity: String,
  targetEntity: String,
  condition: {
    cnt: Integer,
    operator: String,
    device_type: string
  },
  action: String
}
```

Listing 5.2: Data object obtained from rule input form

After the user submits the form an object containing the input values is created as refer listing 5.2. This object is then inspected by the *rule validator*, which provides the method that checks if the values inside the input object are valid according to the grammar described in chapter 4.3.1. If the input is validated by the grammar, then a new *instance of the Rule class* is initialised by passing the corresponding input values and saved into the Liquid Storage. The whole process of rule creation is shown in the sequence diagram in Figure 5.2.

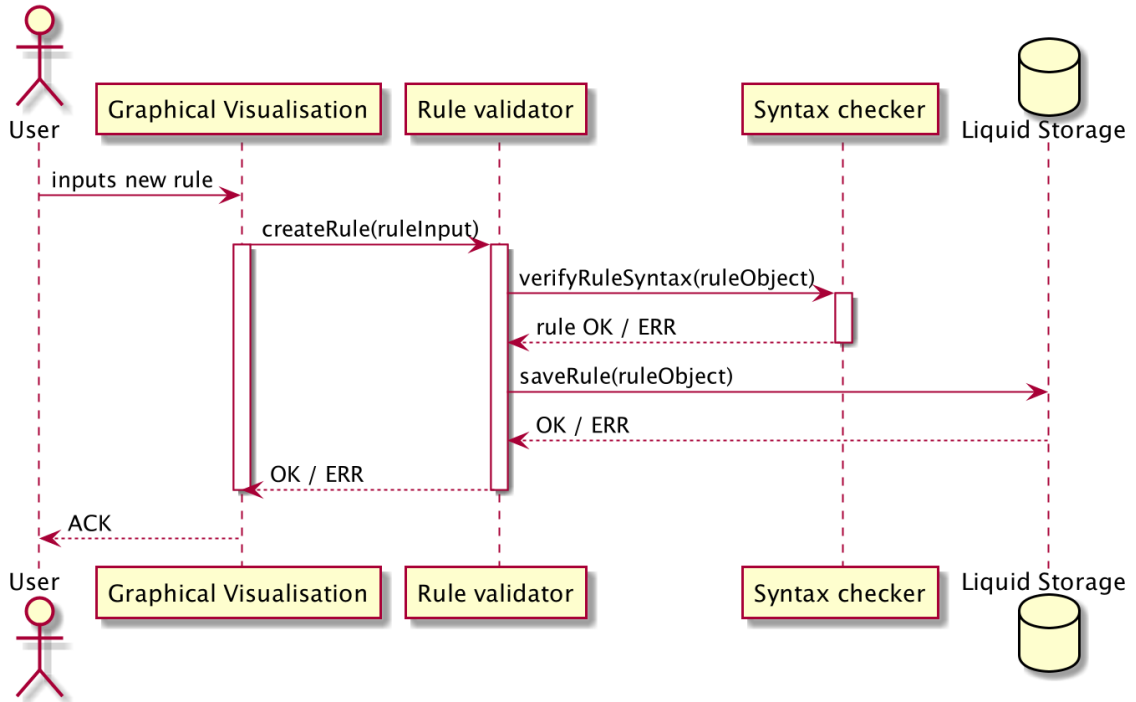


Figure 5.2: Sequence diagram describing the process of rule creation.

5.2.2 Rules storage

As mentioned in the previous section, the existing rules are stored inside Liquid Storage. The rules are saved as an *instance of the class Rule*. The instances of this class define the necessary properties which describe the corresponding rule and the methods that are used for rule manipulation. The class diagram of class Rule is attached in figure 5.3. Inside the Liquid.js Storage, the rule instances are stored into a *Yjs Y.Array*. A newly created rule is appended at the end of the array. The order of the rules in the array is very important because the index in the array also express the priority of the rule. The lower the index is, the lower the priority it has.

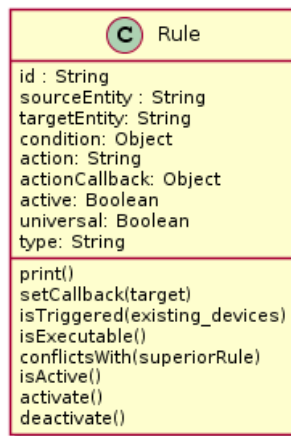


Figure 5.3: Class Rule diagram.

5.2.3 Rule deletion

The user is allowed to delete rules through the UI as described in chapter 4.3.2. When the user deletes any of the existing rules the method linked to this action accesses the HTML list element representing the rule and reads the stored hidden index. This index is later used for accessing the Liquid.js Storage and delete the stored rule object. This process is shown in Figure 5.4.

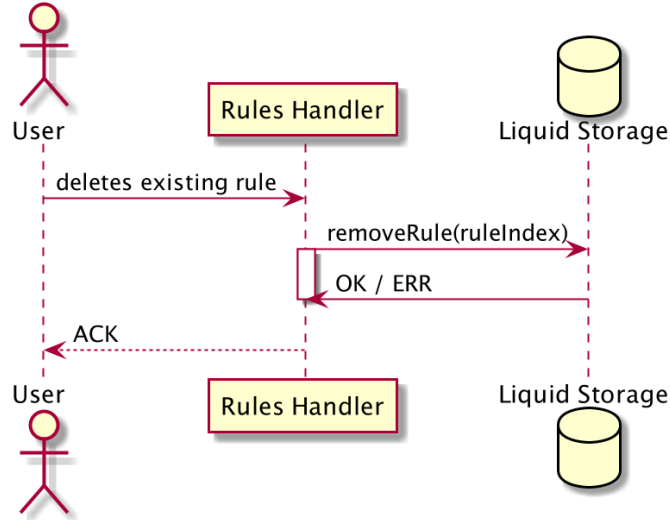


Figure 5.4: Sequence diagram describing the process of rule deletion.

5.2.4 Rule execution

Whenever a new device connects or disconnects, Liquid.js triggers a new event informing all listeners subscribed to the event. Inside the Graph visualisation module, an observer is implemented and listens to these events. Whenever this event is caught, the Graph visualisation informs the Rules Handler module. The rule handler retrieves all existing rules as an array from the Liquid storage and inspects the rule one by one. The rule handler inspects only the active rules and skips all the rules that are deselected by the user in the GUI. First of all the condition of the rule is controlled. For this is implemented rule Class method which compares the current number and types of connected devices with the condition demands. If the condition is evaluated as satisfied, then we need to control the rule action if it is even possible to execute such action in the current application state. To do this, the dedicated method accesses the Cytoscape.js API and reads the current distribution of graph nodes. If the distribution is suitable for the rule action, thus the action is possible, the rule is considered as executed and inserted into the new array dedicated for rules which will be triggered after all of the existing rules will be checked. If in the array are already present any other rules, it is necessary to check if the newly added rule is not in conflict with any of them. To do so, it is necessary to compare the action of newly added action with all the actions of the rules already present in the array. If any conflict is found (i.e. the newly added rule manipulates component which was deleted by any of the previous rules) then the newly added rule is not considered as triggered and is deleted from the array.

After the all the existing rules are checked, for every rule in the final array, the callback of Liquid.js function is prepared for executing the rule action and all the callback functions

are together handed to Graphical Visualisation module which executes the functions one by one. The whole process of finding suitable rules and executing them is visualised as a sequence diagram in figure 5.5.

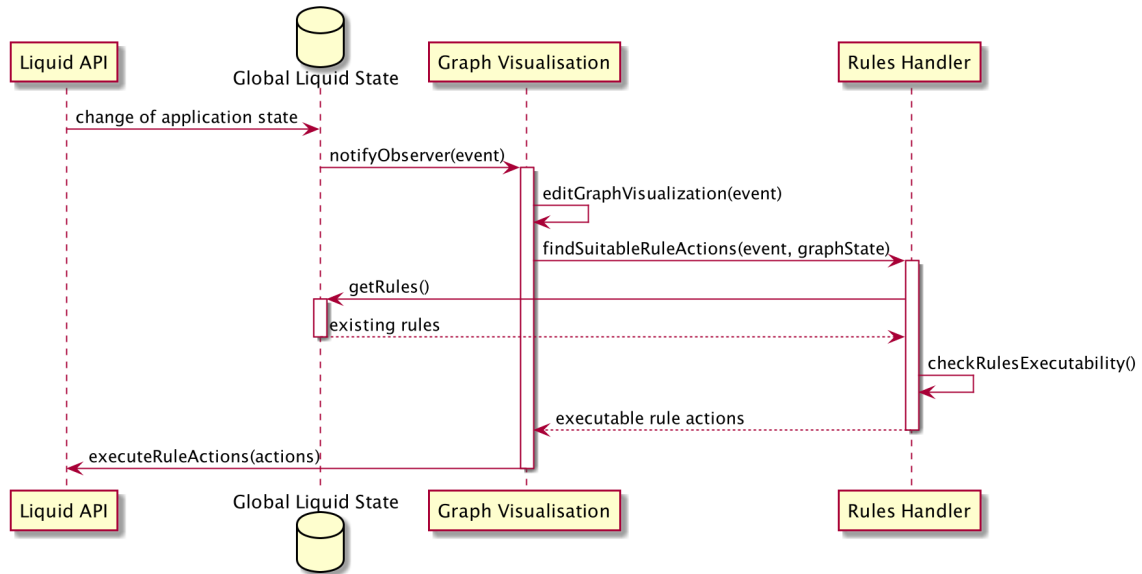


Figure 5.5: Sequence diagram describing the rules execution.

In the case that inside the Liquid.js application would exist two or more complementary view, then all the callback function would be executed multiple times, one time for every existing Complementary view component. To solve this problem, I used the Master/Slave model. There is always maximally one master which executes the rules callbacks. When a new device is connected to the application, it is looking for any other existing device which is labelled as the master. If there is no such device, the newly connected device is proclaimed as the master. Whenever the master device is disconnected, then another component in order is selected as the master. I implemented this whole logic directly into Liquid.js framework and is computed on the server-side of the application.

The whole process of the rule execution was tested. To do so, I implemented the unit test suites testing every method or function involved in the execution process. For the creation of test cases, I used web framework *QUnit*⁵. This framework provides functionality for creation of unit tests inspecting the JavaScript code functionality.

⁵QUnit: A JavaScript Unit Testing framework – <https://qunitjs.com/>

Chapter 6

Demonstration

This chapter demonstrates the final solution of the complementary view. The screen-shot in the figure 6.1 visualise the final graphical interface of the complementary view tool. In the top-left corner, there are manual controls of the graph navigation and zoom. In the bottom-left corner are two buttons opening the UI interface dedicated to rule-based behaviour? In the centre of the image is the most important part, the graph visualisation of the application state. This visualisation consists of the connected devices. First of them is a desktop device running on Mac OS. On this device is instantiated one picture component. This component has two liquid properties: text and image. Both of them are paired with the components present in the second connected device. The second device is an Android smartphone. On the phone are present two components, picture and chat. On the phone the picture component has two liquid properties: text and image. The chat component has one liquid property: history.

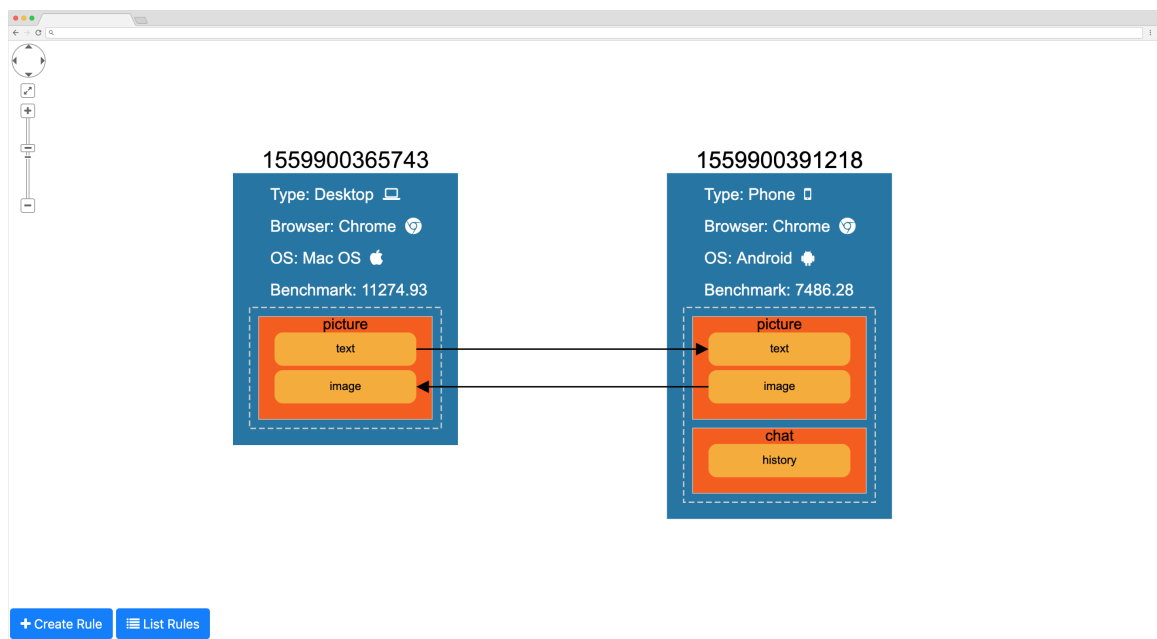


Figure 6.1: Screen-shot of the implemented graphical visualisation displaying application state.

In the screen-shot of figure 6.2 is displayed device node within the graph visualisation. Because this device is highlighted its colour is green. The background of the visualisation is also green because the highlighted device is the one which contains the complementary

view component rendered in this screen-shot. In the screen-shot is also visible a context menu. This menu is opened by right-clicking on the device node and allows to user highlight/unhighlight the device or create new component on that device.

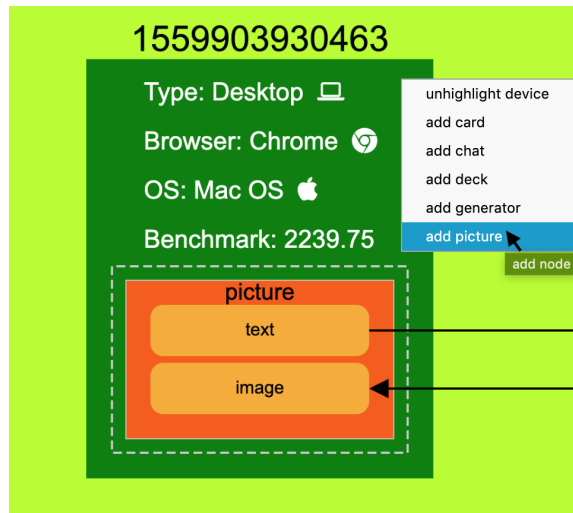


Figure 6.2: Screen-shot displaying highlighted device and controls via context menu.

By clicking on the button *Create Rule* from figure 6.1 the modal window dedicate to new rules definition opens. The screen-shot in figure 6.3 contains this modal window. In this case, the form is already filled. After the clicking on the *Save* button the inputs will be transformed into the rule which will migrate one of the components to any connected tablet if there are more than 2 devices connected within the application.

Source entity	Action	Target entity	Operator	Quantity	State
liquid://155990	Migrate to	TABLET	more than	2	Device(s) exists

Figure 6.3: Screen-shot of the form dedicated for new rules definition.

By clicking on the button *List Rules* from figure 6.1 the modal window dedicate to existing rules management opens. The screen-shot in the figure 6.4 visualise this list. At the moment when the screen-shot was captured, there were three defined rules. Two of the active and one disabled according to check-boxes on the left side of the list. In the screen-shot is visible that user is allowed to change the order (priority) of the rules by drag-and-drop. User is also disabled or activate the rules by provided check-box or delete them by clicking on the trash-bin icon on the right side of the list.

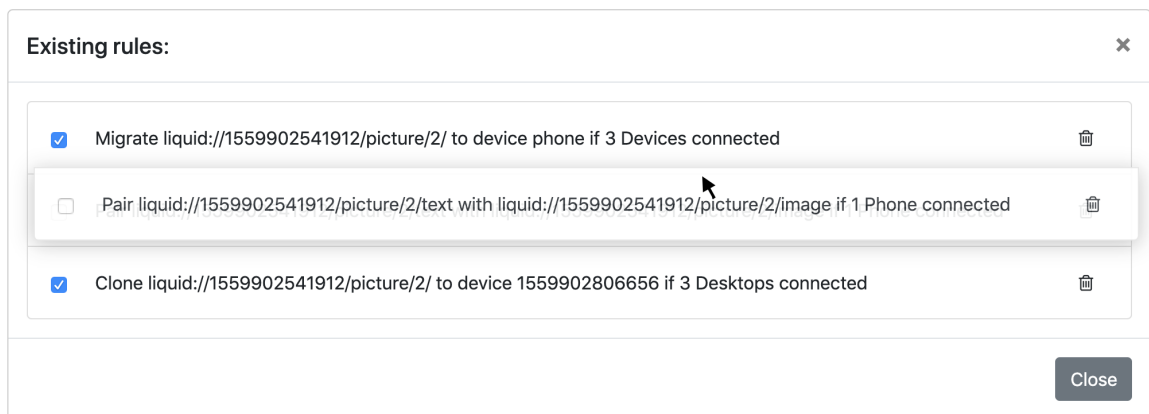


Figure 6.4: Screen-shot of the list displaying existing rules allowing their management.

Chapter 7

Conclusion

The main goal of the thesis was to create a tool Complementary view which extends the functionality of web framework Liquid.js by graphical visualisation of the current state of the application and rule-based behaviour. The introductory part of the text introduces how the Liquid.js framework works and also describes the fundamental idea of Liquid software, the ideal model which Liquid.js builds on. After that, I described the requirements on Complementary view and proposed a solution to fulfilling these requirements. In the text was also described the technical part of the implementation. That includes information about used technologies and programming paradigms. The implementation was successful and the complementary view fulfils all the requirements notice earlier. For the crucial parts of the JavaScript code were created unit tests to check the desired functionality. The final chapter demonstrates the functionality of the final solutions.

7.1 Future work

Although the Complementary view is implemented, there are still ways how to improve the implementation and extend it of new features. Very useful functionality would be the possibility to save the existing rules into the client's local storage. At this moment, if there are existing rules in the application and all the connected devices disconnects, the content of the liquid storage and all the saved rules is lost. This could be solved by serialising the content of the liquid storage and saving it to the file inside the client-side file system. Then this file could be loaded again to fill the liquid storage with its data.

Another useful feature would be displaying the screen-shots of components inside the graph visualisation. Then the user could see not just the basic information about components and their synchronisation, but also their current visual appearance. I tried to implement this feature, but I didn't find any suitable JavaScript library allowing this behaviour and implementing this on my own would exceed the assignment of this project.

Bibliography


- [1] Badam, S. K.; Elmqvist, N.: PolyChrome: A Cross-Device Framework for Collaborative Web Visualization. In *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces*. ITS '14. New York, NY, USA: ACM. 2014. ISBN 978-1-4503-2587-5. pp. 109–118. doi:10.1145/2669485.2669518. Retrieved from: <http://doi.acm.org/10.1145/2669485.2669518>
- [2] Dearman, D.; Pierce, J. S.: „It’s on my other computer!“. *Computing with Multiple Devices*. 2008. ISBN 9781605580111.
- [3] Franz, M.; Lopes, C. T.; Huck, G.; et al.: Cytoscape.js: a graph theory library for visualisation and analysis. *Bioinformatics*. sep 2015: page btv557. ISSN 1367-4803. doi:10.1093/bioinformatics/btv557. Retrieved from: <https://academic.oup.com/bioinformatics/article-lookup/doi/10.1093/bioinformatics/btv557>
- [4] Gallidabino, A.: Liquid.js for Polymer. Retrieved from: <http://liquid.inf.usi.ch/>
- [5] Gallidabino, A.: Migrating and pairing recursive stateful components between multiple devices with Liquid.js for polymer. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2016. ISBN 9783319387901. ISSN 16113349. doi:10.1007/978-3-319-38791-8_47.
- [6] Gallidabino, A.; Pautasso, C.: Deploying Stateful Web Components on Multiple Devices with Liquid.js for Polymer. In *Proceedings - 2016 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering, CBSE 2016*. Venice, Italy. 2016. ISBN 9781509025695. pp. 85–90. doi:10.1109/CBSE.2016.11. Retrieved from: <http://dx.doi.org/10.1145/2872518.2890538>.
- [7] Gallidabino, A.; Pautasso, C.: The Liquid.js Framework for Migrating and Cloning Stateful Web Components across Multiple Devices. In *Proc. of the 25th International World Wide Web conference (WWW 2016)*. Montreal, Canada. 2016. ISBN 9781450341448. pp. 183–186. doi:10.1145/2872518.2890538. Retrieved from: <http://dx.doi.org/10.1145/2872518.2890538>.
- [8] Gallidabino, A.; Pautasso, C.: The Liquid User Experience API. In *The Web Conference (WWW2018)*. Lyon, France. 2018. ISBN 9781450356404. pp. 767–774. doi:10.1145/3184558.3188738. Retrieved from: <https://doi.org/10.1145/3184558.3188738>

- [9] Gallidabino, A.; Pautasso, C.; Ilvonen, V.; et al.: On the Architecture of Liquid Software: Technology Alternatives and Design Space. Technical report. Retrieved from: <http://design.inf.usi.ch/sites/default/files/biblio/2016{~}WICSA{~}Liquid.pdf>
- [10] Gallidabino, A.; Pautasso, C.; Mikkonen, T.; et al.: ARCHITECTING LIQUID SOFTWARE. Technical Report 6. 2017.
- [11] Google: The New Multi-screen World: Understanding Cross-platform Consumer Behavior. 2012. Retrieved from: http://services.google.com/fh/files/misc/multiscreenworld_final.pdf
- [12] Google: The connected consumer. Technical report. 2015. Retrieved from: <http://www.google.com.sg/publicdata/explore?ds=dg8d1eetcqsb1>
- [13] Hartman, J.; Manber, U.; Peterson, L.; et al.: Liquid software: A new paradigm for networked systems. *University of Arizona*. 1996.
- [14] Hartman, J. H.; Bigot, P. A.; Bridges, P. G.; et al.: Joust: A Platform for Liquid Software. *IEEE Computer*. vol. 32. 1998: pp. 50–56.
- [15] Kurose, J.; Ross, K.: *Computer Networking - A top-down approach*. 2005. ISBN 9780133594140.
- [16] Levin, M.: *Designing Multi-Device Experiences: An Ecosystem Approach to User Experiences Across Devices*. 2014. ISBN 0636920027089.
- [17] Marcotte, E.: *Responsive Web Design (Brief Books for People Who Make Websites, No. 4)*. 2011. ISBN 098444257X. 5–9, 44–50, 71–79 pp. Retrieved from: <http://abookapart.com>
- [18] Mikkonen, T.; Systä, K.; Pautasso, C.: Towards liquid web applications. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. vol. 9114. 2015: pp. 134–143. ISSN 16113349. doi:10.1007/978-3-319-19890-3_10.
- [19] Oracle: Sun Ray 3 Series Clients Product Guide. 2012. Retrieved from: https://docs.oracle.com/cd/E25618_02/E25355/E25355.pdf
- [20] Simon, F.; Landman, Y.; Sadogursky, B.: *LIQUID SOFTWARE How to Achieve Trusted Continuous Updates in the DevOps World*. 2018. ISBN 9781981855728.
- [21] Taivalsaari, A.; Mikkonen, T.: ScienceDirect From Apps to Liquid Multi-Device Software. *Procedia Computer Science*. vol. 56. 2015: pp. 34–40. doi:10.1016/j.procs.2015.07.179. Retrieved from: www.sciencedirect.com
- [22] Taivalsaari, A.; Mikkonen, T.; Systä, K.: Liquid software manifesto: The era of multiple device ownership and its implications for software architecture. *Proceedings - International Computer Software and Applications Conference*. 2014: pp. 338–343. ISSN 07303157. doi:10.1109/COMPSAC.2014.56.

- [23] Wikipedia: Client–server model — Wikipedia, The Free Encyclopedia.
<http://en.wikipedia.org/w/index.php?title=Client%E2%80%93server%20model&oldid=905286470>. 2019. [Online; accessed 18-July-2019].
- [24] Wikipedia: Peer-to-peer — Wikipedia, The Free Encyclopedia.
<http://en.wikipedia.org/w/index.php?title=Peer-to-peer&oldid=905757378>. 2019. [Online; accessed 18-July-2019].

Appendix A

Poster



Università della Svizzera italiana


Complementary View Adaptation with LIQUID.JS

Student: Petr Knetl Advisor: Prof. Cesare Pautasso Co-advisor: Andrea Gallidabino

Motivation

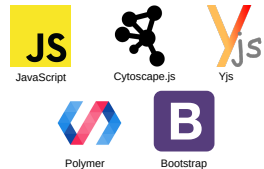
The goal of the project is the creation of a complementary view tool for Liquid.js which dynamically visualize the state of the user interface of liquid applications as an oriented graph. Moreover it implements an additional interface that allows to apply rule-based complementary view behaviours to liquid components.

Liquid.js basics



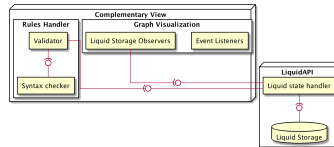
- Divides the web application into components and allows to distribute them among multiple devices
- Creates p2p communication channels between every pair of connected devices
- The components can be moved from one device to another
- The components contains properties (variables) which can be synchronized with properties of any other existing component

Technologies



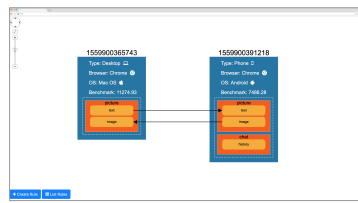
JavaScript Cytoscape.js Yjs Polymer Bootstrap

Design



- Divided into two cooperating modules: Graph visualization and Rules Handler
- Component data shared among devices via global Liquid Storage implemented as Yjs Map

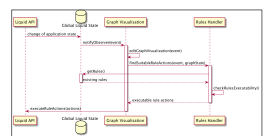
Graph visualization



- Displays the basic information about devices and current deployment state of the liquid application as oriented graph
- Allows user modify the state through the GUI
- Implemented with use of Cytoscape.js

Rule-based behaviour

- User can predefine reactions to specific state of the liquid application
- The user defined rules are composed of action and condition. The rule action is triggered when defined condition is met
- Possible actions are migrate, fork, clone component and pair, unpair property
- Condition describes the desired state of application which triggers the rule action, example: "if at least one tablet device is connected"



Future work

possibility to save the existing rules

- Currently lost if all devices disconnects
- Serialize the current state and save in the file
- Implement functionality to overwrite current state by the file content

screen-shots of components in the visualization

- Extends the displayed information about component
- Allows user to see all the components in one view

Figure A.1: Poster created for thesis defense at Università della Svizzera italiana.

Appendix B

CD Contents

On the attached CD are files organised as follows¹:

```
/
├── src
│   ├── public
│   │   └── cmpViewResources - implementation of The Complementary View
├── doc - thesis text and poster
├── lib - implementation of Liquid.js
├── liquidScripts - implementation of Liquid.js
├── config.js
└── README.md
```

¹For simplicity, only important files are mentioned.

Appendix C

Installation manual

Since the Liquid.js needs a running server, to run the application on the local computer, it is necessary to install some tools. The following commands must be run in order to successfully install and run the application:

1. Download and install **Node.js**¹ (minimum version v6.6.0), **npm**² (minimum version v3.10.9) and **MongoDB**³ (in this order⁴).
2. Start MongoDB database by running command `mongod` in terminal.
3. in the root directory run command `npm start` to start the local server.
4. Visit `localhost:8888/cmpView.html` to open session with Complementary view component and `localhost:8888` to open session without any component.
5. (Optional) To be able to connect other devices through the local network, modify the `liquidConfig` variable in `complementaryView/public/index.html` and `complementaryView/public/cmpView.html` files so the `host` property contains local IP address of the machine running the server. Then to access the page proceed same as point 4 refers, just with the machine IP instead of `localhost`.
6. (Optional) In order to run unit tests, in the `complementaryView/public/index.html` uncomment block including tests and access `localhost:8888`.

Implementation of The Complementary view is located in `/src/public/cmpViewResources` directory.

¹Node.js: <https://nodejs.org/>

²npm: <https://www.npmjs.com/>

³MongoDB: <https://www.mongodb.com/>

⁴Detailed installation guide of these tools can be found on their websites