Expertise
and insight
for the future

Junjie Yin

Salesforce Lightning - Usability of Lightning Web Components

Metropolia University of Applied Sciences

Bachelor of Engineering

Software Engineering

Bachelor's Thesis

30.03.2019

Author Title Number of Pages Date	Junjie Yin Salesforce Lightning - Usability of Lightning Web Components 31 pages 23 Nov 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructor	Auvo Häkkinen, Principal Lecturer

The topic of the study is the Salesforce new programming model: Lightning Web Component and its usability. The study also compares the new model with the original model, the Aura component and Lightning Web Component.

All the developments were done in the Salesforce platform which is the most used Customer Relationship Management (CRM) platform in the world. The development works were done by using Intellij IDEA and Visual Studio Code as the development tools.

The thesis is divided into two parts: theoretical and practical. The theoretical part briefly covers the Salesforce architecture, Lightning frameworks, and Apex. The practical part concentrates on how to build and test and application using the Lightning Web Components and Aura components. The paper discusses the results and problems faced during the development work.

As a result of the study, the research question as to the usability of Lightning Web Components was answered thanks to the development of the application. The shortcomings of Lightning Web Components are specifically pointed out at the end of the thesis.

Keywords	Salesforce, CRM, Lightning Web Components, Lightning Aura Components

Tekijä Otsikko Sivumäärä Aika	Junjie Yin Salesforce Lightning - Usability of Lightning Web Components 31 sivua 23. 11. 2019
Tutkinto	Insinööri(AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaina	Ohjelmistotuotanto
Ohjaajat	Yliopettaja Auvo Häkkinen

Insinöörityössä on tavoitteena soveltaa uutta Salesforcen ohjelmointimallia, Lightning Web Components. Työn tarkoitus on tutkia ohjelmointimallin käyttökelpoisuus, sekä hyvät ja huonot puolit vertailemalla edelliseen Lightning Aura Component – malliin.

Toteutusympäristönä oli Salesforce, joka on mailman käytetyin asiakkuudenhalluntajärjestelmä(CRM). Kehitystyökaluna on käytössä sekä Intellij IDEA että Visual Studio Code. Kaikki ohjelmointi kehitykset on tehty Salesforcen Developer Edition -kehitysympäristössä.

Projekti on jaettu kahteen osaan: teoreettinen osuus ja käytännöllinen osuus. Teoreettisessa osuudessa selitään lyhyesti Salesforcen perusosa-alueista: Salesforcen tietorakenne, Lightning Aura Components, Lightning Web Components ja Apex. Käytännöllisessä osuudessa syvennytään ohjelmointimalleihin, missä kehitetään kaksi sovellusta käyttäen uutta ja vanhaa ohjelmointimallia. Samalla käsitellään myös Salesforcen ohjelmointimallien hyviä käytäntöjä. Lopussa vertaillaan tuloksia molempien mallien kesken.

Insinöörityön tuloksien perusteella voidaan antaa jonkinlainen lopputulos, eli onko uuden Salesforcen ohjelmointimallin käyttökelpoinen. Tuloksessa mainitaan erityisesti Lightning Web Componentin kehityksen aikana esitetyistä rajoituksista.

Avainsanat	Salesforce, CRM, Lightning Web Components, Lightning Aura Components

Contents

List of Abbreviations

1 Introduction 1

2 Salesforce Environment 2

2.1 Salesforce Architecture 2
2.2 Lightning Aura Components 3
2.3 Lightning Web Component 5
2.4 Apex 8

3 Research Approach 9

3.1 Building Aura Component 10
3.1.1 Salesforce Lightning Event 13
3.1.2 Lightning Aura Components Testing and Result 15
3.2 Building Lightning Web Component 17
3.2.1 Lightning Web Component Event 23
3.2.2 Lightning Web Component Testing and Result 24
3.3 Results 26

4 Conclusions 27

References 29

List of Abbreviations

AI	Artificial intelligence is intelligence demonstrated by machines.
CRM	Customer Relationship Management. A way to manage a company's relationships with customers and potential customers.
DML	Data Manipulation Language, used for inserting, deleting or updating the data in a database.
LWC	Lightning Web Component, new programming model for the Lightning component.
SOQL	Salesforce Object Query Language, similar to Structured Query Language (SQL), but only used in Salesforce environment
ORG	a.k.a., organization – a single client of the Salesforce application.
SOSL	Salesforce Object Search Language, construct text-based queries that allow users to search based on for example text, email or phone numbers
UI	User Interface in which a user may interact from an information device.

1 Introduction

Cloud computing delivers different kinds of computing services to end-users such as software, databases, and services, over the internet. Nowadays, cloud computing is commonly used in daily lives. Anyone online uses the cloud. Sharing documents via drives, sending emails or posting photos on social media are made possible by cloud-based computing.

CRM (Customer Relationship Management) might refer to technology, a strategy or a process. All these are for managing a company's relationships with (potential) customers. Salesforce is one of the most used platforms in the world which combines both CRM and cloud computing. In 2019, Salesforce released a new programming model—Lightning Web Components—which has a different framework compared to the original model, Aura Component.

The goal of this thesis was to investigate the usability of the Salesforce Lightning Web Component. This thesis is divided into two parts: theoretical and practical. The theoretical part briefly covers what Salesforce, Lightning frameworks, and Apex are. The practical part concentrates on how to build and test the Lightning Web Components and Aura components. The best practices of Lightning Components are summarized also in the practical parts. How components communicate with each other and communicate with the Apex controller are mentioned in this part. The purpose of this part is to describe how to build two components with similar functionality using two different models.

At the end of the thesis, benefits or limitations arising from the new model are compared with the original model. The thesis also analyzes the usability of the new model from a business point of view. What are the possibilities to use the new model in real business projects at Fluido Oy? What are the present limitations in using Lightning Web Components? The thesis aims to give a good overview of the new Lightning model without requiring previous understanding of the Lightning framework.

2 Salesforce Environment

Salesforce.com is an American cloud-based software company. Salesforce provides a platform that helps companies understand their customers' needs and solve problems by managing customer information and interactions in a better way. It is the customer success platform helping companies connect to their customers, prospects, and partners by making easy-to-use business applications.

The practical parts of the present study are fully built in the Salesforce environment. Salesforce has its own coding languages and frameworks, whose details are covered in the next chapters.

2.1 Salesforce Architecture

Many people know salesforce from CRM and source offerings. Salesforce also offers a platform as a service that allows users to build their own custom applications in a very efficient way. The platform provides a rich set of capabilities and features that can help users to build custom applications. The Salesforce platform structure is divided into five different areas: Multitenant infrastructure, Data Platform, Einstein which is an AI (Artificial intelligence) platform, Developer platform, and Salesforce applications. Figure 1 illustrates the most common way to show how Salesforce architecture is built.

Figure 1. Salesforce architecture [2].

One can imagine Figure 1 as a big building where each floor sits on top of each other. Everything inside of the building is powered by metadata and made up of different parts, for example, data service and AI. On the roof, users can see all the applications that Salesforce developers have built on top of the platform. Salesforce highlights that everything has resided in the trusted, multitenant cloud.

Even if trust is the top priority at Salesforce, multitenancy is what makes cloud computing a real possibity. That is why it is the second word that Salesforce highlights. For the multitenancy, Salesforce uses the apartment building as an image to explain how it works. Each company has its own room in the cloud or in the building, but they are not alone. No matter the size of the company, they share and use the same resource. Trust and multitenancy go side by side, and Salesforce is the key player in the house, to keep each company data secure even when they are sharing spaces.

2.2 Lightning Aura Components

Lightning Aura Components is the original programming model for building Lightning components. Aura components are self-contained and reusable units of an application. The framework contains a lot of prebuilt components that developers can summon during development. For example, in the Lightning Design System, styling is available in the Lightning namespace. The difficult part of prebuilt components is customization. Every customer wants different functionality, it is hard to use prebuilt components for every customer. Depending on the needs, the prebuilt components are not always the best options. No matter how complicated the application is, the basic structure is always the same. Figure 2 shows the basic structure of the Lightning Component.

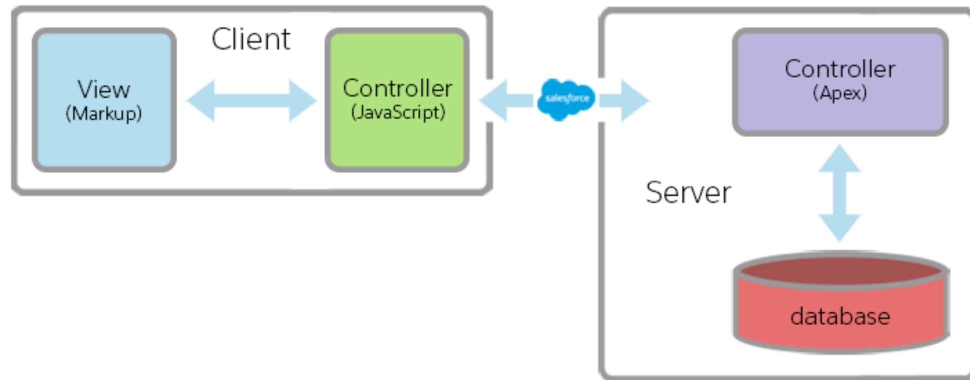

Figure 2. Lightning Component Basic Structure [10].

Figure 2 presents the Lightning Component basic structure, demonstrating how it communicates between the server and client-side in a Salesforce environment. The View is basically the UI the user will see in the Org. Under the view, there are JavaScript controllers and helpers. Controllers play a key role in the Lightning Components, which talk to the server-side. On the server-side, the Apex controller or trigger takes the biggest role. The user can call the Salesforce database using the Apex Controllers.

When building a lightning component, it is always good to consider using the standard components before considering creating a custom. It will make development easier and faster regardless of the limitations of the standard components. Figure 3 presents a simple Lightning Component which includes a standard Lightning input component.

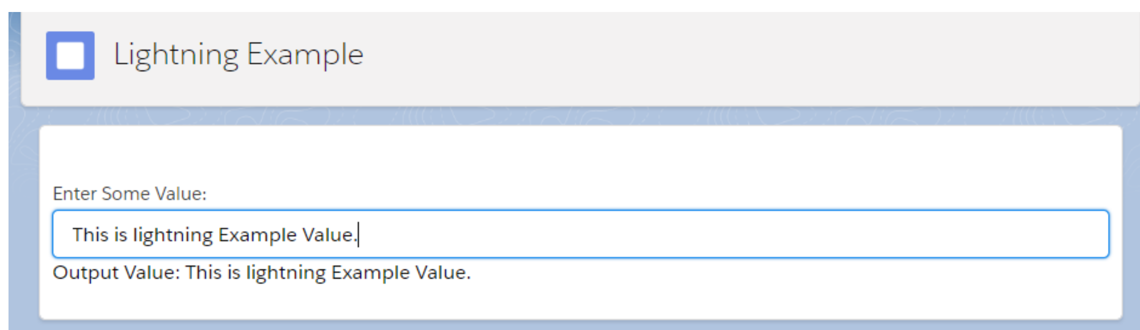

Figure 3. Simple example of Aura Component.

The user can input a certain value inside the input field and the value will be populated synchronously in the Output Value area. The code is very simple and clean using the prebuilt standard component (Code example 1).

```
<aura:attribute name="exampleValue" type="String" />
<div class="slds">
    <lightning:card class="slds-p-around_x-small">
        <lightning:input type="text" label="Enter Some Value: " value="{!v.ex-
ampleValue}" />
        <ui:outputText value="{!'Output Value: ' + v.exampleValue}"  />
    </lightning:card>
</div>
```

Code example 1. Simple prebuilt Aura Component example

As showns in Code example 1, developers are able to do this with only a few lines of code. Aura attributes are variable where the value will be saved after the user has an input and putting the same attribute as the outputText value will make the value populate synchronously. Aura attributes can be different types, for example, strings, lists or even sObjects. In Salesforce.com, any object either Standard or Custom is an sObject. An sObject is generic and it can be used to represent any object type e.g. Account and Location. In Code example 1, the component only has a string type attribute to save the input value. More details about Aura Attributes and Expressions are discussed below.

2.3 Lightning Web Component

Salesforce has released a new programming model for building Lightning components, Lightning Web Components. The feature was released in the Spring'19 release. After Salesforce had released the Lightning Component framework in 2014, many web stack features that required framework at that time are now standardized. That is also the reason why Salesforce wanted to develop the Lightning Web Components, to bring the latest improvements in JavaScript and web standards to the Lightning Component framework. Lightning Web Components are based on modern web standards as Figure 4 presents.

Figure 4. Lightning Web Components is built on web standards [9].

Figure 4 shows, Lightning Web Components is built on top of the Web Standards features and uses core Web Components standards. Because of this, it can be powered up by ES6+, classes, modules, custom elements, templates and more. What Lightning Web Components provide on top of the Web Standards is the latest of specialized Salesforce service. The biggest benefit of the new programming model is exceptional performance and only provides the necessary parts to make everything work well in all browsers. Figure 5 shows how the 2014 web stack is built.

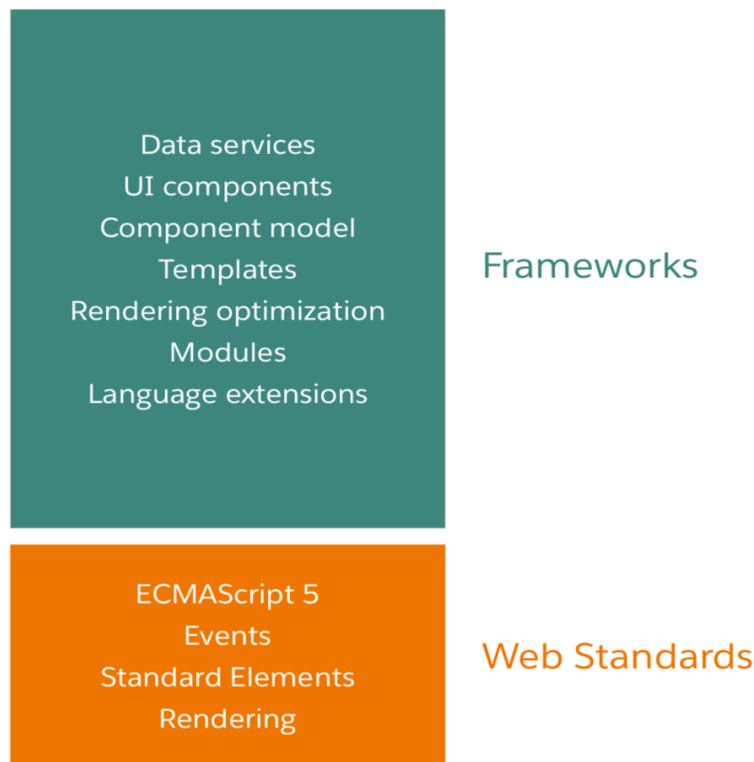

Figure 5 The 2014 Web Stack [9].

As mentioned before, the Web Standards only offered a limited foundation of full-stack. The core language was ECMAScript 5, and standard elements did not support any custom elements. These are just a few examples of the limitations that 2014 Web Standards has brought to developers. The cost for that is that developers or software vendors need to jump in to fill the gaps. It is also one of the reasons why Salesforce launched the Lightning Component framework in 2014. It came with its own component model and its own modular programming model. The web stack for 2019 looks totally different in Figure 6.

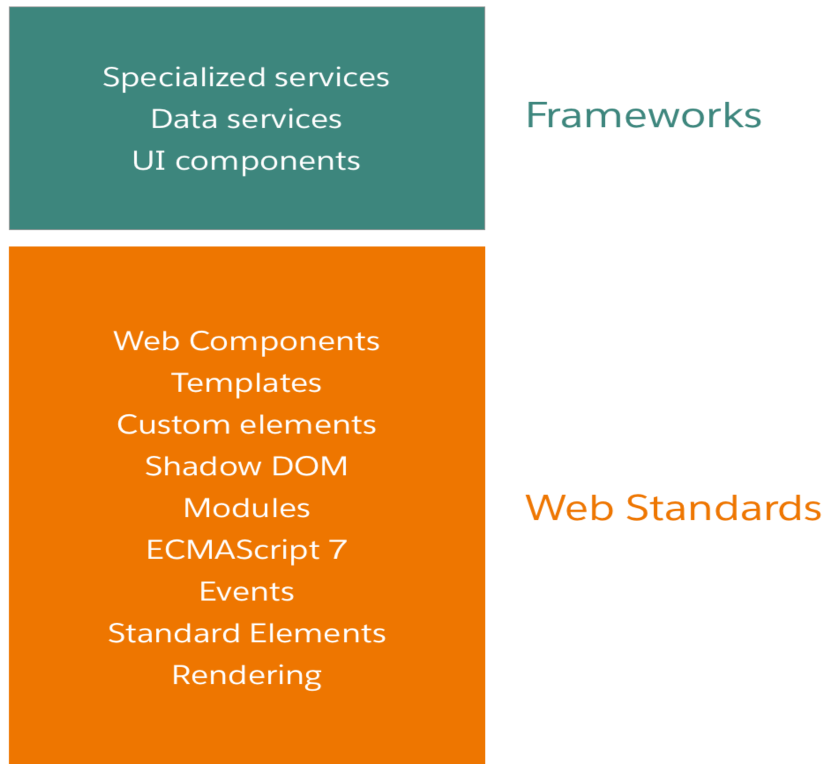

Figure 6. The 2019 web stack [9].

Nowadays, the web stack looks different compared to the 2014 web stack. The 2019 web stack is more focused on standards. Many features that used to require frameworks now become standard, for example, Web Components and a lot of Custom elements were included in the Web Standards. Neither a proprietary component model nor proprietary extensions are necessary to the developers. That means no more filling the gaps. Frameworks can focus on providing a thin layer of specialized service on top of the standard stack.

2.4 Apex

Apex is a Salesforce object-oriented programming language based on familiar Java idioms, such as variable and expression syntax. Apex allows developers to build a connection to Salesforce.com's back-end database and client-server interface. Apex provides a lot of built-in support, for example, DML (Data Manipulation Language) calls, such as

insert, update and delete data from the database. SOQL (Salesforce Object Query Language) and SOSL (Salesforce Object Search Language) queries that return lists of sObject records.

An Apex class can also be tied to objects by using Apex Triggers. Developers use triggers when DML call is affecting another record or need some validations. Apex triggers can be executed before or after the DML calls.

3 Research Approach

As discussed in the previous chapter regarding the basic structure and theory of Lightning Aura Component and Lightning Web Components, the goal of the project were: 1) to build two similar components or applications, one using the Lightning Aura Component and the other using Lightning Web Components; 2) to discover the difference between the two programming models; 3) to illustrate the process of building the component using the best practice for each programming model; and 4) to highlight the pros and cons of each model.

The initial design was to include as many functionalities as possible for both applications using a different model. The application was to have several components. An event was built to make components communicate with each other. Figure 7 represents the mockup for the application

Search

Search

OPPORTUNITY NAME	ACCOUNT NAME	CLOSE DATE	STAGE	CONFIDENCE	AMOUNT	CONTACT
Cloudhub	Cloudhub	4/14/2018	Prospecting	20%	$25k	jrogers@cloudhub.com
Cloudhub + Anypoint Connectors	Cloudhub	4/14/2018	Prospecting	40%	$30k	jrogers@cloudhub.com

Figure 7. The application Mockup for Initial Design. The mock-up is built with Justinmind app.

. The idea was to build an application that has at least four different components. The first component contains everything inside of the application as the main component. The second component is a component for the filter, which includes one or more input fields and a search button. The third component is the table component. It contains a data table, showing the date from the Salesforce database. Each row in the table is the fourth component. It is a child component for the table component. The reason, why the table was separated into two different components, is because it makes the code cleaner and easier to read. For the custom component, it was more straightforward to customize and reuse the component later on. Each row has different functionality depending on what kind of attribute developer is passing from the parent component, and in this case, it is the table component.

3.1 Building Aura Component

Starting from the beginning, Salesforce already has its own rules for naming the Lightning components. When a component is created, the name must begin with a letter; it cannot contain anything else except alphanumeric and underscore characters; the name cannot be duplicated [3]. Other than these rules, every customer's org (Organization) can have their own rules or best practices for naming the Lightning components. Figure 8 shows how the naming works for the table Lightning components.

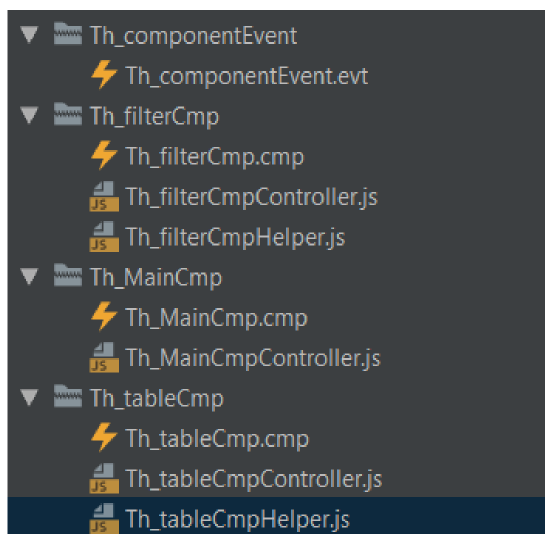

Figure 8. How to name the components.

All the related components start with ‘Th’, it is a short version from the ‘Thesis’ word. This will help developers to recognize that all components start with Th are related to the same application. It might not be so useful when one has only one or two applications in the org, but when developers are working for a bigger org with tens of different applications and components, this will help a lot in finding the correct components.

Each component contains three main files inside of the Aura Component bundle. The file ends with .cmp is the Component markup. Most of HTML tags can be used in the markup, such as <div> or <p>. In the bundle, it has two JavaScript files, JavaScript Controller and helper. Both JavaScript files might look the same, but they have different meanings and play different roles in the component. The Controller works only for the component markup inside of the component bundle, but the Helper can be shared across everything. It will allow developers to reuse the method in the Helper to avoid unnecessary duplicated codes to keep logic clean and more understandable. Everything starts from the main Component, which has been most often called Parent Component, as Code example 2 shows.//Component Markup

```
<aura:component description="Th_MainCmp"
                implements="force:appHostable,flexipage:available-
ForAllPageTypes "
                controller="Th_cmpController"
                access="global">
    <aura:attribute name="listOfContact" type="List" />
    <aura:handler name="ComponentEvent" event="c:Th_componentEvent" ac-
tion="{!c.handleComponentEvent}"/>
    <aura:handler name="init" value="{!this}" action="{!c.doInit}"/>
    <div class="slds">
        <c:Th_filterCmp />
        <c:Th_tableCmp contactList="{!v.listOfContact}"/>
    </div>

</aura:component>
```

Code example 2, The main Lightning Component Markup.

If the application contains a lot of functionality or pages, it is always better to separate them into different child components. In Code example 2, the Main component contains two child components, one is for the filter and the other one is for the data table. Code example 3 shows the JavaScript controller and helper class for the main component.

```
// Component JavaScript Controller
```

```
({
    doInit : function (component, event, helper) {
        helper.getData(component, event ,helper);
    },

    handleComponentEvent : function (component, event, helper) {
        var filteredList = event.getParam("filteredList");
        component.set('v.listOfContact', filteredList);
    }
})

//Component Helper
({
    getData : function (component ,event, helper) {
        var action = component.get("c.initData");
        action.setCallback(this, function (response) {
            var state = response.getState();
            console.log('State: ', state);
            if(state === "SUCCESS"){
                var responseCtrl = response.getReturnValue();
                console.log('Contact List data: '. responseCtrl);
                component.set("v.listOfContact", responseCtrl.contactList);
            }
        });
        $A.enqueueAction(action);
    }
})
```

Code example 3. The main Lightning component JavaScript Controller and Helper.

When a user opens the application from the Org. The Init handler will automatically be fired and then the application will be initialized, to get all the data needed from the server-side. The handler is handled by a client-side JavaScript controller as Code example 3 shows.

From the server-side Apex controller, a connection to the Salesforce Database is built, for example, DML calls. There are some rules to communicate with Lightning Aura Component. As Code example 4 shows, the structure of the Apex controller is close to Java as mentioned in the previous chapter.

```
public with sharing class Th_cmpController {
    @AuraEnabled public Contact[] filteredList;
    @AuraEnabled public Contact[] contactList;
    @AuraEnabled public Boolean isError = false;
    @AuraEnabled public Boolean isAlert = false;
    @AuraEnabled public String message = '';


    @AuraEnabled
```

```
    public static Th_cmpController initData (){
        Th_cmpController ctrl = new Th_cmpController();
        try{
            ctrl.contactList = [SELECT Id, Name, Email, Birthdate, AccountId,
Account.Name FROM Contact];
            System.debug('search: ' + ctrl.contactList);
        }catch(Exception e){
            handleExceptions(true, e, ctrl);
        }

        return ctrl;
    }
}
```

Code example 4. Apex controller for the Table App.

In Code example 4, there is a lot of @AuraEnabled mark before the variable or method. Before methods mean these methods can be called from the JavaScript Controller. Before variables mean these variables can be returned to the JavaScript Controller.

3.1.1 Salesforce Lightning Event

How the components communicate with each other? The communications are handled by events. Salesforce provides two types of events: Component event and Application event. Salesforce recommends using the Component events before considering using the Application events. The communication is simpler and easy to handle using the Component events [4].

In the example shown in Figure 8, the Th_componentEvent.evt file handles the communication between components in the application. In Code example 5, the Component Event is simple.

```
<aura:event type="COMPONENT" description="Th_componentEvent">
    <aura:attribute type="List" name="filteredList" />
</aura:event>
```

Code Example 5. The Component Event.

The event contains only one line of code. It is an attribute that will store the value that the child component has fired from the JavaScript Controller or helper. It is important that the attribute has the correct type. In this case, the event will store the list of contact after

the user used the filter. From the FilterCmp component JavaScript Controller, the event will be fired as above Code example 6 shows

```
//Child component Markup
<aura:registerEvent name="ComponentEvent" type="c:Th_componentEvent"/>

//Child component
fireEvent : function (component ,event, newList) {
    var eventCmp = component.getEvent('ComponentEvent');
    console.log('fire...', newList);
    eventCmp.setParams({"filteredList" : newList});
    eventCmp.fire();
}
```

Code Example 6. How to fire the Component Event.

. First, it will get the event by using the name value, which is registered in the component Markup. The next step is to set up the parameters for the event. The last step will be when the cmpEvent.fire() is executed. That means the event is fired.

A handler is needed in the parent component to handle the attribute that is coming from the event. In the parent component markup, <aura:handler/> defines which event the component is handling and what kind of action will be used to handle the event from the JavaScript Controller side. It must be the same-named events and use the same event class with the child component [5]. Code examples 2 and 3 show how the parent component handles the event in the application.

The value or attribute can be passed only from a child component to a parent component by using Component events [4]. The Lightning events are an imponent part of the Lightning framework and the relationship between components is not always so simple. That is why the Application events are needed. Firing an Application event is similar to the Component event, but all components that have a handler with the same event class are notified. Figure 9 shows the basic architecture for both event types.

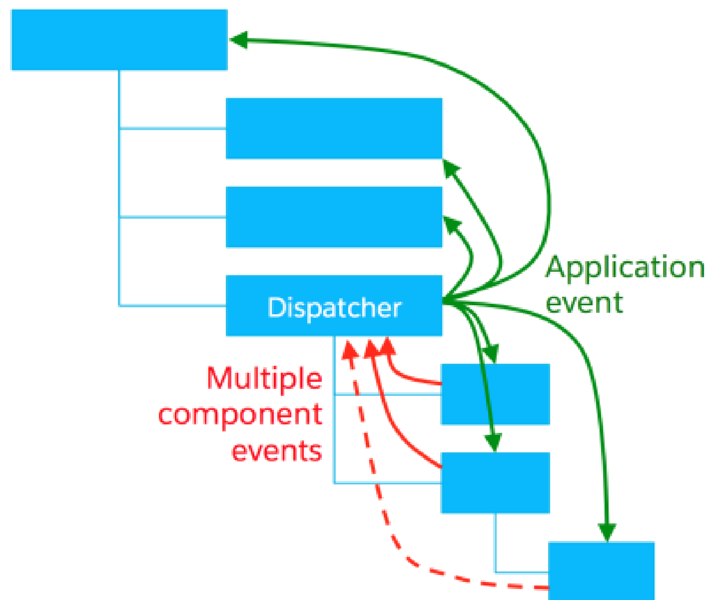

Figure 9. Lightning Event architecture [7].

As mentioned before, the best practice would consider always using a component event before thinking about the application event. However, if facing a blocking use case or a complex architecture, an application event might be a better way to handle the events [7].

3.1.2 Lightning Aura Components Testing and Result

Testing the Lightning Aura Components is always a little challenging. Salesforce does not provide any official tool or best practices the developers could follow and do the unit tests for the Lightning Aura Component. Usually, how the components are tested is so that the developers can run the component or application in the Organization and see if it is working as expected. Figure 10 shows how the application is tested from UI.

Search By Name

Rose

Search

Contact Name	Account	Email
Rose Gonzalez	Edge Communications	

Figure 10. Testing the Contact Table App from UI.

First, some value is put in the filter field and then it is seen if the table shows what is expected. The user has put 'Rose' as the searching value for the name. In the table, it only shows the contact that has 'Rose' in Contact Name. So, the Contact table application is working as expected.

Testing the Apex logic is somehow more important for Salesforce then testing the component itself. Every time when trying to deploy Apex classes from sandbox to another, all the Apex class unit test coverage needs to be over 75%. Otherwise, the deployment will fail. In Code Example 7, the unit test is made for the Apex class.

```
@IsTest
private class Th_cmpControllerTest {
    @IsTest
    static void initDataTest() {
        Contact[] contacts = [SELECT Id, Name, Email, Birthdate, AccountId,
Account.Name FROM Contact];

        Test.startTest();
        Th_cmpController ctrl = Th_cmpController.initData();
        Test.stopTest();
        System.assertEquals(contacts.size(), ctrl.contactList.size());
    }
}
```

Code Example 7. Apex Unit test class.

@IsTest defines that the class contains a unit test class and that the method is a unit test method. The test will pass if the Apex method is called in the test method and does not return any errors, but that is not enough for the best practice. A good test should always compare the returning value with the expected value that has set up before the test. In this case, the unit test is checking if both list sizes are the same. Figure 11 shows the final results of the application.

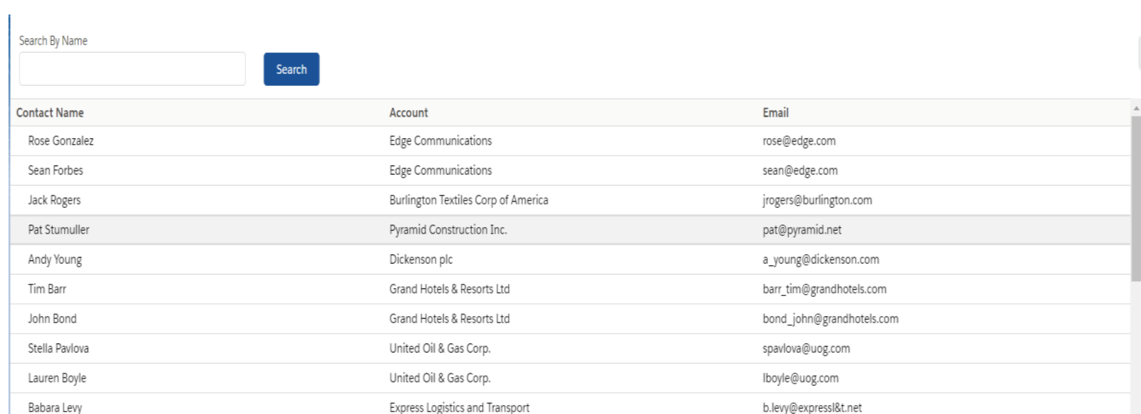

Figure 11. Table App UI.

No critical difficulties or problems were met during building the application. Everything that has been designed in the beginning was able to finalize as showing in Figure 11. The application was able to include most of the basic stuff about Lightning Aura Components. In the next session, a similar application needs to be done using the Lightning Web Component.

3.2 Building Lightning Web Component

For building the Lightning Web component, it was decided to use the Visual Studio Code instead of Intellij as the development tool. The reason for this is that the Visual Studio Code has a good extension called Lightning Web Components which provides code-editing features for the Lightning Web Components.

In the Lightning Web Component, the design model is like the Lightning Aura Component. The application has the main component which contains several child components. The naming for the Web Component is a little different compared to Aura Component and has more rules for the naming, for example, files names must begin with a lowercase letter and only with alphanumeric or underscore characters [8]. Figure 12 shows, what Lightning Web Component bundle contains.

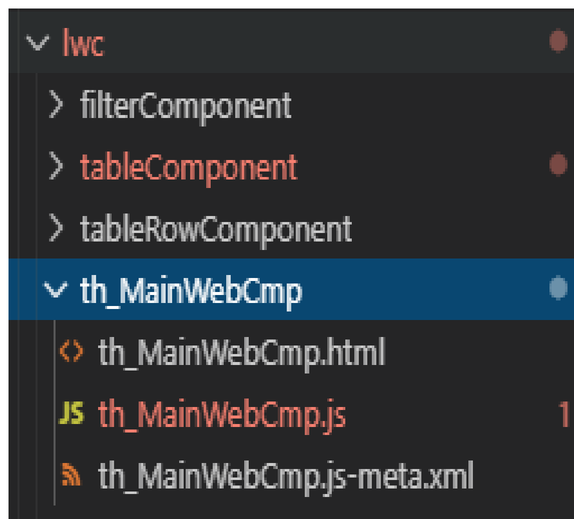

Figure 12. Lightning web components file and naming.

A Lightning Web Component must contain three different files: an HTML file, a JavaScript file, and a Component Configuration file. HTML provides the UI for the component. JavaScript defines the core logic and handling events. The configuration file defines the metadata values for the component, for example, the design configuration for the Lightning App Builder [9]. A CSS file can be also included in the component bundle, but it is optional. Code Example 8 shows an example of how the configuration could look like.

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>47.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__RecordPage</target>
        <target>lightning__HomePage</target>
      </targets>
</LightningComponentBundle>
```

Code example 8. The main component configuration file.

In this case, the configuration file defines where the application can be shown.

As mentioned before, the structure is more or less the same as in the Aura component. Everything starts from the Main component which has two different child components. The only difference is, the whole table is in the same component instead of separating them into two different components. The reason for that is covered in the following parts. Code example 9 shows that the Main Component contains two child components, and everything is wrapped in the HTML template tag.

```
<template>
 <div>
   <lightning-card  title="Contact Table" icon-name="action:new_contact">
    <c-filter-component onfilterthelist={handleTheList}></c-filter-component>
    <c-table-component contactlist={contactList} filtered ={filtered}></c-ta-
ble-component>
   </lightning-card>
 </div>
</template>
```

Code example 9. The Main Web component HTML file.

The first one is for the filter and the second one is for the table. The way of containing the child component is pretty much the same as in the Aura component, with <c-child-component></c-childcomponent>html tag instead of <c:childcomponent>. Depending on

Metropolia
University of Applied Sciences

the need, different attributes can be passed inside of the tag to the child component. In this case, the contact list is passing from the parent component to the child component. The template looks very similar to Aura templates. The biggest change would be that no need to declare variables in the template. Instead, everything will be declared in the JavaScript file. This change reduces a huge amount of line for the HTML file.

All the business logic is defined in the JavaScript files. Compare to Aura Components, Lightning Web Components use common JavaScript ECMAScript 8 methods and syntax. Code example 10 shows an example of how JavaScript file can be built in the Lightning Web Components.

```
import { LightningElement, wire, track } from 'lwc';
import { getListUi } from 'lightning/uiListApi';
import CONTACT_OBJECT from '@salesforce/schema/Contact';
import getContact from '@salesforce/apex/lwcTableCtrl.getContact';

export default class th_MainWebCmp extends LightningElement {

   @track contactList;
   @track searchValue;
   @track filtered = false;

   @wire (getContact, { searchName: '$searchValue'})
   filterContacts ({error, data}){
       if(data){
          /* eslint-disable no-console*/
           console.log('Filtered List: ' + JSON.stringify(data));
           this.contactList = data.filteredList;
           this.filtered = true;
       }else if(error){
           this.error = error;
           this.filteredList = undefined;
       }
   }

   @wire(getListUi, { objectApiName: CONTACT_OBJECT, listViewApiName: 'AllCon-
tacts'})

   wiredContactList({
      error, data
   }){
      if(data){
      this.contactList = data.records.records;
      /* eslint-disable no-console*/
      console.log('Contact list: ' + JSON.stringify(this.contactList));
      }else if(error){
     //Handle the error
         this.error = error;
      }
   }

 handleTheList(event){
    //get the value from the event
    this.searchValue = event.detail;
```

```
    /* eslint-disable no-console*/
    console.log('Parent Search Value:' +  this.searchValue);

 }

}
```

Code example 10. The Main Component JavaScript file.

The JavaScript file must include the export statement which defines a class extends the LightningElement class which is imported from the built-in module for Lightning Web Components. In the code example 10, the import statement indicated the JavaScript uses the LightningElement, track and wire functionality from the 'lwc' module.

Decorators such as track and wire are used in JavaScript to extend the behavior of a class, property or method [11]. Most commonly used Lightning Web Component decorators are:

- @api

  The api decorator defines property as public reactive. It means the property can be used in the template or other components. For example, when trying to pass a value from a parent component to a child component, in the child component, the property needs to mark with the api decorator.

- @track

  The track decorator is private reactive. It is marked for internal monitoring. Every time when marked property's value has changed, it will force a component to rerender. It is very useful especially when a user interacts with the component. In the code example 10, the searchvalue is marked with track decorator which means every time when the value has changed, the getContact method will be rerendered again to get correct data from Salesforce database.

- @wire

  The wire decorator simplifies getting and binding data from a Salesforce organization. In the code example 10, the getContact method is called with only one

line of code. Lightning Web Components also provide some wire adapter for making development easier and let's delve into the adapter in the next part.

A new feature of Lightning Web Component is prebuilt adapters to fetch data from the database without creating an Apex controller. In code example 10, getListUi is one of the adapters that can be used in the JavaScript controller to get the records and metadata from a list view. First, one passes the object API name, and that is going to be the contact object in this case. The second parameter is the list view API name. The wire adapter is currently in beta [12]. So the feature still has a lot of limitations for getting the data. For example, the adapter will not accept any conditions as a parameter. Of course, a new list view can be created, but it will not be the best practice from a development perspective for a real business case. That is also the reason, the Apex class is built to gather the contacts after the filtering. After the data is returned from the database, it will pass to the child table component as Code example 11 shows below.

```
<template>
    <table class="slds-table slds-table_cell-buffer slds-table_bordered">
        <thead>
            <tr class="slds-line-height_reset">
                <th class="" scope="col">
                    <div class="slds-truncate" title="Name">Contact Name</div>
                </th>
                <th class="" scope="col">
                    <div class="slds-truncate" title="Account Name">Ac-
count Name</div>
                </th>
                <th class="" scope="col">
                    <div class="slds-truncate" title="Email">Email</div>
                </th>
                <th class="" scope="col">
                    <div class="slds-truncate" title="Phone">Phone</div>
                </th>
            </tr>
        </thead>
        <tbody>
            <template for:each={contactlist} for:item="con">
               <template if:false={filtered}>
               <tr class="slds-hint-parent" key={con.fields.Id}>
                    <td  data-label="Name">{con.fields.Name.value}</td>
                    <td data-label="Account Name">{con.fields.Account.dis-
playValue}</td>
                    <td data-label="Email"> {con.fields.Phone.value}</td>
                    <td data-label="Phone"> {con.fields.Email.value}</td>
                </tr>
            </template>

            <template if:true={filtered}>
                <tr class="slds-hint-parent" key={con.Id}>
                     <td  data-label="Name">{con.Name}</td>
                     <td data-label="Account Name">{con.Account.Name}</td>
                     <td data-label="Email"> {con.Phone}</td>
```

```
                    <td data-label="Phone"> {con.Email}</td>
                </tr>
            </template>

           </template>
       </tbody>
   </table>
</template>
```

Code example 11. Child table component HTML file.

Lightning Web Component has HTML template directives which used to handle by HTML Aura tags in Lightning Aura Component. The basic behaviors are the same. The HTML template for:each directive iterates through the Array that the child component gets from the parent component and displays the fields in the table. In Code example 11, there are two different <tr> tags inside of for:each directive to show the fields. The reason for it is because the returned lists are different from the Apex controller or directly using the wire adapter, as Figure 13 shows.


```
Contactlist: [
  {
    "apiName": "Contact",
    "childRelationships": {

    },
    "fields": {
      "Account": {
        "displayValue": "Dickenson plc",
        "value": {
          "apiName": "Account",
          "childRelationships": {

          },
          "fields": {
            "CreatedDate": {
              "displayValue": null,
              "value": "2018-07-10T10:48:58.000Z"
            },
            "Id": {
              "displayValue": null,
              "value": "0011t000002ZjitAAC"
            },
            "LastModifiedById": {
              "displayValue": null,
              "value": "0051t0000001227AAA"
            },
            "LastModifiedDate": {
              "displayValue": null,
              "value": "2018-10-17T14:50:20.000Z"
            },
            "Name": {
              "displayValue": null,
              "value": "Dickenson plc"
            },
            "SystemModstamp": {
              "displayValue": null,
              "value": "2018-10-17T14:50:20.000Z"
            }
          },
```

```
Contact: {
  Id=0031t0000020nedAAA,
  AccountId=0011t000002ZjiqAAC,
  CreatedDate=2018-07-1010: 48: 58
},
```


Figure 13. Returned JSON list from the adapter(left) and Apex controller(right).

From an adapter, all the fields are wrapped in a fields property that has a name. To get the actual value, HTML needs to call <fields.field_name.value>. From the Apex controller, simply just call the field name from the list.

A JavaScript file is simple in the child table component. Code example 12 shows how everything can be handled only with a few lines of code.

```
import { LightningElement, track, api } from 'lwc';

export default class tableComponent extends LightningElement {
    @api contactlist;
    @api filtered;
}
```

Code example 12. Child table component JavaScript file.

As mentioned before, the api decorator defines the property value as public reactive. It is not enough just passing the value from the parent component to the child using the HTML file. What if the child component needs to fire a value back to the parent component? Lightning Web component also provides an event that allows doing that. Let us dive into the Lightning Web Component Event in the next section.

3.2.1 Lightning Web Component Event

Lightning Web component dispatches standard DOM events. It is also possible to implement custom events. Custom events are used to communicate up from a child component to a parent component. The events are built on DOM events. Salesforce recommends using the CustomEvent interface instead of the Event interface for a more consistent experience across browsers [13].

Different from the Aura component, there is no need for creating a Lightning Event file to handle the event between two-component. Everything can be handled in the HTML and JavaScript file. Use the CustomEvent() constructor to create an event and dispatchEvent() method to fire an event, as Code example 14 shows.

```
export default class filterComponent extends LightningElement {
    @track searchValue = '';
    @track filteredList;
    @track error;
```

```
    changeHandler(event) {
        const searchKey = event.target.value;
        // eslint-disable-next-line @lwc/lwc/no-async-operation
        if(searchKey.length > 2 || searchKey.length === 0){
            this.searchValue = searchKey;
            /* eslint-disable no-console*/
            console.log('Search value: ' + this.searchValue);
            //Create the Custom Event
            const fireTheValue = new CustomEvent('filterthelist', {de-
tail: this.searchValue});
            //dispatch the Event
            this.dispatchEvent(fireTheValue);
        }
    }

}
```

Code example 14. Filter component JavaScript file where an event is created.

A required parameter is needed when creating a custom event, which is the event type. The event type can be named with any string, but need to conform with the DOM event standard, with no uppercase letters, spaces, and use underscores to separate words. The second parameter is the data that passes in the event. In this case, the searched value from the input is passed to the parent component.

The Lightning Web Component Event can be handled in two ways: expressively from the component's HTML template, or adding the handler programmatically using a JavaScript API [13]. In this case, it is handled in the HTML template. In the HTML template, the child component needs to use an HTML attribute with syntax onEventtype. In Code example 9, the filter child component has a onfilterthelist attribute. In the JavaScript class, the handleTheEvent method will be executed when the event is fired as Code example 10 shows.

3.2.2 Lightning Web Component Testing and Result

Lightning Web Components provide the capability to do the unit tests. This is a huge improvement for the Lightning Components because the code should always be tested as well as possible. Jest is the testing framework for JavaScript and it also provides features for testing the Lightning Web Components.

Every test added to the project represents a subfolder for the Lightning Web Component. In Figure 14, the unit test is in the __tests__ folder under the main component bundle.

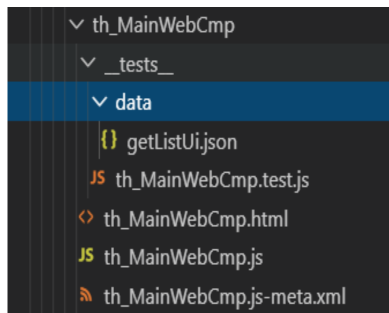

Figure 14. Unit test class located in the __test__ subfolder under the component.

Under the test folder, there is another subfolder which is a data folder. The folder contains the JSON file which is a mockup data for the unit tests as Code example 15 shows. The JSON can be custom, but it might get some errors during the testing. The Salesforce recommended to take a snapshot of the data by accessing the UI API. In this case, a snapshot was taken from the console log and formatted into the JSON as Figure 13 shows.

```
import { createElement } from 'lwc';
import { registerLdsTestWireAdapter } from '@salesforce/sfdx-lwc-jest';
import Main from 'c/th_MainWebCmp';
import { getListUi } from 'lightning/uiListApi';

// import data from the mockup Json file
const mockGetListUIData = require('./data/getListUi.json');

const getListUiWireAdapter = registerLdsTestWireAdapter(getListUi);

describe('c-th_MainWebCmp @wire demonstration test', ()=> {
    while(document.body.firstChild){
        document.body.removeChild(document.body.firstChild);
    }


    it('render the th_MainWebCmp with mock up data', ()=>{
            const element = createElement('c-th_MainWebCmp',{is: Main});
            document.body.appendChild(element);
            getListUiWireAdapter.emit(mockGetListUIData);

            return Promise.resolve().then(() => {
            });
    });
});
```

Code example 15. How the unit test class looks like for the Lightning Web Components.

Metropolia
University of Applied Sciences

In Code example 15, all the basic setups are done for the unit testing. The mock data should be imported before the test and controlled using the registerLdsTestWireAdapter() method, the test wire adapter under the test. This is how the mock data passed into the tests.The adapter should be always reset by calling removechild method before each test. So, the next test will not be affected by previous tests [15].

Even though Lightning Web Components provide a unit test functionality using the Jest testing tool, testing in the UI is also important. The Lightning Web Components application looks a bit different compared to the Aura components as Figure 15 shows.

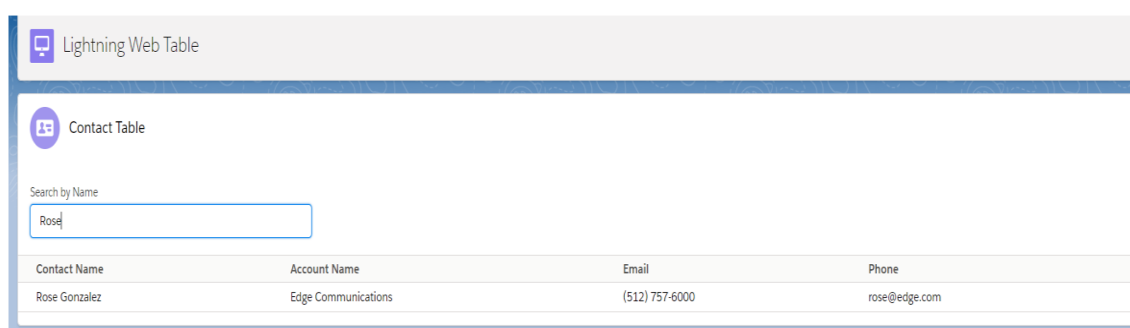

Figure 15. Testing the filter functionality from the App.

When compared with the application built using the Lightning Aura Components shown in Figure 10, the application looks slightly different and the filter also works differently, even the result after the filtering is the same. In this application, there is no search button. How the filter works, is whenever the search value length is over three, the search method will be fired, and the contacts list will be returned from the Apex class. The reason for removing the button will be explained in the next chapter.

3.3 Results

Both components were built successfully. The differences and features are introduced inthe previous chapters. The whole application was built within 230 lines of code using Lightning Aura Components and 165 lines of code using Lightning Web Components without including the test classes. The size of the application has reduced significantly mostly because of the Lightning Web Components adopter and decorators. Even the

Lightning Web Components decorator that is used in the application is still the beta version but the functionality is very powerful for getting the data without writing an Apex class.

The Aura components model is more mature after all the years of development. Even the model contains a lot of custom functionality, but it still covered mostly what is needed during the development. It can be noticed while building the application because the development went smoothly without any critical issues.

The same does not happen for the Lightning Web Component. During the development, there were a lot of issues that came up on the way. The biggest issue faced was the styling for the components, as mentioned in the earlier sections about putting the whole table in the same component and removing the button from the user interface. That is all because of the problem with the CSS styling and the reason for the problems is the shadow DOM. Shadow DOM is the internal document object model (DOM) [14]. It is a web standard feature that will be automatically created for the Lightning Web Components. After the implementation, it brings a lot of limitations to the CSS, for example, only affecting the top-level of prebuilt Lightning components like Lightning buttons which is totally different from Aura components, which the styling also can affect to the internal components.

The way of building the application using both models in this study can also be developed in many different ways. Especially, the Lightning Web Components is still a new programming model and new features are coming out in every release. The best practice for the new programming model is changing all the time during every release.

4 Conclusions

Overall the project has achieved what was planned in the beginning. The aim of the project was to investigate the usability of the Lightning Web Components during the development, pointing out the good and bad points of the new model.

From a technical point of view, the Lightning Web Component model is still too immature so that even some component catching problems were faced during the development. Besides, Salesforce provides good documentation about the model to make it easier for developers to get hands-on it but because of the limitations and bugs, sometimes it would be better still going with the old programming model.

In Fluido Oy, the main developments are still done with the Lightning Aura Components. Many of the developers did not even consider using the new programming model. From a business point of view, it will take too much time to learn how to build applications with the new model. Especially, if the developer is already highly skilled with the Lightning Aura Components. Even after the learning, the developer never knows what kind of issues will be faced during the development and the practical documentations are still so limited from the internet that sometimes it is hard to find a solution to the problem.

For the conclusion, considering all the powerful features and functionalities that the model has brought with it, Lightning Web Components are the future of Salesforce development, but it will take some time before it becomes as mature as Aura components currently are.

References

1 Learning Salesforce Coding with example [online]. <http://sfdcinpractice.com/index.php/2017/02/22/2-2-explanation-lightning-component-works/> Accessed October 2019.

2 Understand the Salesforce Archtecture [online]. <https://trailhead.salesforce.com/en/content/learn/modules/starting_force_com/starting_understanding_arch> Accessed October 2019.

3 Lightning Aura Components Developer Guide:Component Names [online]. <https://developer.salesforce.com/docs/atlas.enus.lightning.meta/lightning/components_names.htm> Accessed November 2019.

4 Salesforce Lightning Events [online]. <https://www.mstsolutions.com/technical/salesforce-lightning-events/> Accessed November 2019.

5 Understanding Lightning Components events naming and parameters[online]. < https://medium.com/@amster/understanding-lightning-components-events-naming-and-parameters-297db051af79> Accessed November 2019.

6 Application Events[online]. < https://developer.salesforce.com/docs/atlas.en-us.lightning.meta/lightning/events_application.htm> Accessed November 2019.

7 Lightning Inter-Component Communication Patterns[online]. < https://developer.salesforce.com/blogs/developer-relations/2017/04/lightning-inter-component-communication-patterns.html> Accessed November 2019.

8 Lightning Web Components Reference[online]. < https://lwc.dev/guide/reference> Accessed November 2019.

9 Introducing Lighning Web Components, Developer Guide[online]. <https://developer.salesforce.com/docs/component-library/documentation/lwc> Accessed November 2019.

10 Connect to Salesforce with Server-Side Controllers[online]. < https://trailhead.salesforce.com/en/content/learn/modules/lex_dev_lc_basics/lex_dev_lc_basics_server> Accessed November 2019.

11 Create Lightning Web Components[online]. < https://trailhead.salesforce.com/content/learn/modules/lightning-web-components-basics/create-lightning-web-components> Accessed November 2019.

12 LWC – getListUi Beta. <https://wipdeveloper.com/lwc-getlistui-beta/ > Accessed November 2019.

13 Lightning Web Component Events[online]. < https://lwc.dev/guide/events> Accessed November 2019.

14 Shadow DOM [online]. < https://developer.salesforce.com/docs/component-library/documentation/lwc/lwc.create_dom> Accessed November 2019

15 Write Jest Tests for Lightning Web Components [online]. <https://developer.salesforce.com/docs/component-library/documentation/lwc/lwc.unit_testing_using_jest_create_tests> Accessed November 2019.