Asema IoT Central
Component Driven Programming with React and QML Manual

ASEMA

Table of Contents

1. Introduction

React is one of the most popular web front end frameworks at the time of writing this document. Unsurprisingly, it is also a prime candidate for writing web interfaces for IoT applications. QML on the other hand is a declarative language for interface driven applications. Built similarly on the concept of components, QML is a great tool for building UI's for platform native applications.

The purpose of this manual is to show with the help of coding examples how these two technologies are similar, where they differ and how the two can be used for user interfaces when working with Asema IoT software. The primary focus of this document is to help in switching between those technologies. So if you are skilled in React, how would you go about using those skills with QML. Or if you have QML code, how would you use that to build a React application.

Note
Note that this document is not a detailed programming guide to Asema IoT, but focuses on the differences and similarities of the abovementioned technologies only. For full instructions on how to program the features of Asema IoT into a UI, please refer to the separate Asema IoT UX Programmer Manual. It has plenty of examples in both of these technologies also.

React focuses on dividing the UI into various components and uses JavaScript/JSX) and CSS as its base technologies. QML has its own declarative syntax although the logic programming is done in JavaScript. The design philosophies of these two technologies, namely defining components and declaring their styles, makes switching between the two straightforward if you've followed the practice of properly defining components, as recommended by both technologies.

In Asema IoT, the so called screenlets which run as widgets on Asema IoT Central desktop interface and Asema IoT Dashboard mobile interface exclusively use QML as their technology. Asema IoT web applications on the other hand can use various web technologies, including React, Angular, Vue, and jQuery. So depending on the eventual application architecture you are aiming for, you may find yourself working with all of the technologies at the same time.

This document illustrates on the similarities between designing UI in QML and React to facilitate QML developers in re-implementing their screenlets as web apps with React or vice versa. More information about React and it's features can be found from the official React documentation at https://reactjs.org/. Similarly, more information about QML can be found in QML official documentation at https://doc.qt.io/qt-5/qmlapplications.html

1.1. React Components

React components represent an isolated piece of a user interface. The components are written in JSX. React components are able to contain and manage their own state in various properties. A simple react component is described in the snippet below

```
import React, { Component } from 'react'
class ReactComponent extends Component {
    render() {
        return(
            <h1 classname="reactHeader">This is a React component</h1>
        )
    }
}
```

The simple React component above renders a h1 html element when called.

1.1.1. State and Props

React components can contain states of various properties. They can manipulate their state and propagate their state through props. The example below shows the use of state and props.

```
class StateProps extends Component {
    constructor(props) {
        super(props)
        this.state = { clicked: false }
    }
    render(){
        <h1>{this.props.title}</h1>
        <p>Clicked: {this.state.clicked} </p>
    }
}
```

The above snippet shows the use of state and props. Further information about state, props and their use and manipulation can be found in the official React documentation.

1.2. QML Components

Similar to React, QML components also represent an isolated piece of a user interface. The components are written in the QML markup language with a syntax somewhat similar to JSON. QML components also have their own state in various properties. A simple QML is shown below

```
import QtQuick 2.3
Item {
    Text {
        text: "This is a QML component"
    }
}
```

The sample QML component above renders a text element to the top of the screen with default style (black text).

1.2.1. States and properties

QML components can also have properties that retain their values and automatically react and re-evaluate when something changes. Re-evaluation takes place automatically when the value of a property changes and it can also be controlled with the signal/slot structure (i.e. a value change is explicitly sent to various other components by emitting a signal of a change). QML also contains states and can have state change rules that dynamically change the UI when a state changes. The example below shows the use of properties in QML.

```
import QtQuick 2.3
Item {
    property string title: "Some title"
    property bool clicked: false

    Column {
        Text {
            text: title
        }
        Text {
            text: clicked ? "Clicked" : "Not clicked"
        }
    }
}
```

2. QML vs React in actual applications

Though different technologies, QML and React are both component based UI design tools making them quite similar to work with. Some of the main similarities and differences between QML and React are listed below:

- QML and React are both component based. In React components are created by coding new subclasses of a Component class while in QML the components are created by naming a declarative tag and writing the tags that create the UI within it.

- QML and React both use JavaScript for application logic. React is "full" JavaScript and everything except for stylesheets is done with it while QML uses JavaScript snippets that are embedded inside the declarative tags (although in QML it is also possible to instantiate components with pure JavaScript)

- React is a JavaScript library that needs a JavaScript engine and a browser whereas QML is a declarative language that runs within Qt applications.

- The styles and formatting in QML is done in the component description where as in React developement stylesheets (eg. CSS, LESS) are used.

- QML provides a set of pre-built components such as Rectangle and Buttons where as React components are created primarily with JSX. QML components can also be created and interfaced with C++.

Because of the similarities in design philosophy, it is easy to structure the applications to mimic one another. Just create components with pretty much the same names and properties and handle the application logic with similar JavaScript code where necessary. If you know the logic of one application in one technology, it is quite straigtforward to find the flow of a translated application in the other technology. Of course the two technologies do not share the same syntax so the code needs to be rewritten and you need to be familiar with the syntax of both.

Because of the component driven design, the core structure of both React and QML applications looks very compact. And it is so. However, once styling is applied to the user interface, this is where the differences start to get highlighted. In QML, the styling and positioning of components is achieved by setting the styling properties of the elements inside the QML code. React on the other hand uses the battle tested CSS styling that separates style from declaration. This makes the core code of QML longer than that of React.

QML has been designed ground up for modern style one page applications and uses a widget layout that is usually much easier to manage than applications that must rely on browser rendering engine and the CSS box model. For example, the struggle of positioning an element at tbe bottom of the page tends to be and eternal problem for web coders (including those that use React) whereas in QML it is a trivial positioning of an anchor. Then again, React has the support of thousands of libraries and tools for making all sorts of weird and wonderful UI features while such extensions are limited in QML. Doing AJAX to a backend may also be somewhat easier with React, though QML also has pretty good support for it.

2.1. Comparison of QML and React component rendering

To show how QML and React compare in a real application, in this section, we will create a QML file and it's corresponding React component. The example is based on Ice Cream Stand, a full example application that you can download from Asema IoT downloads section. Loading this example and studying it should give you a good understanding on how to apply both technologies to an application UI.

In this particular example, we'll take one component of this Ice Cream Stand application, the Brand component. In the application, a Brand renders and represents on product (an ice cream sold at the stand) and styles it according to the brand of the product and the application. The React equivalent of this component is then shown.

2.1.1. QML version of the example component

The Brand.qml file is presented in the snippet below.

```
import QtQuick 2.3
import QtQuick.Controls 2.0
import QtQuick.Layouts 1.0
import QtGraphicalEffects 1.0
import Jarvis 1.0

Rectangle {
    id: iceCreamBrand
    width: 210
    height: 210
    radius: 6
    border.width: 1
    border.color: "#36576d"
    color: "transparent"

    property string imageSrc
    property variant product: null

    signal selected(string name, string description, double price);

    layer.enabled: true

    layer.effect: DropShadow {
        horizontalOffset: 0
        verticalOffset: 0
        radius: 8.0
        samples: 16
        color: Qt.rgba(0, 0, 0, 0.4)
        source: iceCreamBrand
        transparentBorder: true
    }

    Column {
        width: parent.width
        height: parent.height
        spacing: 6

        Item {
            height: 16
            width: 100
        }

        Image {
            id: pic
            source: (product != null) ? "file://" + deviceContext.getMetaImagePath() + product.meta.picture :  ""
            height: 100
            width: 100
            anchors.horizontalCenter: parent.horizontalCenter
       }

        Item {
            height: 8
            width: 100
        }

        Text {
            id: brand
            text: (product != null) ? product.name : ""
            Layout.fillWidth: true
            wrapMode: Text.WordWrap
            color: "#fffbe8"
            width: parent.width
            horizontalAlignment: Text.AlignHCenter
            font.family: "Elektra Text Pro"
            font.pointSize: 16
            font.bold: true
        }

        Text {
            id: details
            text: (product != null) ? product.description : ""
            wrapMode: Text.WordWrap
            color: "white"
            Layout.alignment: Qt.AlignHCenter
            width: parent.width
            horizontalAlignment: Text.AlignHCenter
            font.family: "Elektra Text Pro"
            font.pointSize: 12
        }
    }

    Rectangle {
        id: productPriceDisplay
        color: "red"
        width: 60
        height: 60
        radius: 30
        anchors.top: parent.top
        anchors.right: parent.right
        anchors.topMargin: 0
        anchors.rightMargin: -8

        Text {
            anchors.centerIn: parent
            text: (product != null) ? product.cost.toFixed(1) + " €" : ""
            color: "white"
            font.family: "Elektra Text Pro"
            font.pointSize: 15
            font.bold: true
        }
    }

    MouseArea {
        anchors.fill: parent
        onClicked: {
            if (product != null)
                iceCreamBrand.selected(product.name, product.description, product.cost);
        }
    }
}
```

Let's analyze a bit what is happening in this component. First, the product is presented as a rectangle that the buyer of an ice cream can click on to buy the product. This is the Rectangle with id "iceCream-Brand". At the very end of the file is the MouseArea that activates when the rectangle is clicked.

The Rectangle has some styling in the form of a drop shadow and then the graphical elements inside of it: a Column with its first element acting as a simple spacer, then a product image followed by the product name and the description. A null check is applied to all of them so that if an errorneous product object is given to the component the errors are handled gracefully.

The product image is fetched from the backend system with a URI which is assembled from the base URI (`deviceContext.getMetaImagePath()`) and picture path stored in the object metadata (`product.meta.picture`). Finally, the product price is displayed in "productPriceDisplay" as a text inside a circle.

When the product is clicked, the component uses the QML signaling methods to send a signal upwards to the application logic. This takes place in the MouseArea by emitting the signal that is defined as `signal selected(string name, string description, double price);`.

The data that this component renders is defined in the property `product`. This property should be set by the UI code when the component is rendered and placed on screen.

2.1.2. React version of the example component

Now, how would we go about doing the same in React? Let's first take a look at the code. First the JSX code and then the CSS are shown below.

```
import React, { Component } from 'react';
import './Brand.css';

class Brand extends Component {

    constructor(props) {
        super(props);
        this.state = {}
        this.onBrandClicked = this.onBrandClicked.bind(this);
    }

    onBrandClicked(event) {
        this.props.selectedCallback(this.props.product.name(), this.props.product.description(), this.props.product.cost());
    }

    render() {
        const isrc = "/iot/meta/" + this.props.product.gid() + /picture/;
        return (
            <div className="Brand" onClick={ this.onBrandClicked }>
                <img src={isrc} className="productimage"></img>
                <span className="productname">{ this.props.product.name() }</span><br/>
                <span className="productdescription">{ this.props.product.description() }</span>
                <div className="productcost">{ this.props.product.cost().toFixed(1) } &euro;</div>
            </div>
        );
    }
}
```

```
export default Brand;
```

```
.Brand {
 border: 1px solid #36576d;
 border-radius: 5px;
 width: 210px;
 height: 210px;
 text-align: center;
 position: relative;
 margin-bottom: 68px;
 margin-left: 68px;
 box-shadow: inset 0px 1px 5px #717171;
 display: inline-block;
 cursor: pointer;
}

.Brand .productimage {
 width: 100px;
 height: 100px;
 display: block;
 margin: 0px auto;
 margin-bottom: 16px;
 margin-top: 19px;
}

.Brand .productname {
 color: #fff;
 font-weight: bold;
 font-size: 1.2em;
 margin-bottom: 7px;
 text-shadow: 0px 1px 5px #666;
}

.Brand .productdescription {
 color: #fff;
 font-size: 1.1em;
 text-shadow: 0px 1px 5px #666;
}

.Brand .productcost {
 position: absolute;
 top: -5px;
 right: -5px;
 background-color: #ff0000;
 width: 58px;
 height: 58px;
 border-radius: 29px;
 color: #fff;
 line-height: 58px;
 font-weight: bold;
 font-size: 1.2em;
 border: 1px solid #9f0000;
}
```

Now you'll see how separating the CSS from the JSX makes the logic code much more compact, although the combined length is pretty close to the same.

As in QML, in React we are doing a similar rectangle with a slight drop shadow effect and then filling it with an image, a product name and a description. The code in `render()` is very compact as all it needs to do is to output the HTML tags and the CSS then styles them.

Similar to QML, the product from where the data is rendered is stored in properties (`this.props.product`). As React does not support similar signaling methods as QML, a callback is used to convey the information about purchase. (`this.onBrandClicked`).

2.1.3. Comparison

The QML and React components described in above snippets yield the same user interface component. However, the difference in styles and description are well pronounced.

- QML components are created from existing primitive UI building blocks (eg. Rectangle) whereas React components are created with HTML like tags.
- In QML the styling of the elements and their attributes are defined in the element itself (e.g. the height, the width and position of a element say Column are defined within its definition) whereas React supports the use of stylesheets (`import './Brand.css'`).

- The onClicked event in QML is defined as a JavaScript within the MouseArea and propagated as signals whereas in React, the event directly triggers JavaScript callback function.

2.2. Data model and component reuse comparison

Now that we know how to draw one item, let's next see how to apply that knowledge to a more commonplace scenario where there is data of several items that all need to be represented and rendered by some user interface component. In this chapter, we'll be focusing on utilising the components we created above to output values from a data model.

2.2.1. QML version of component reuse

First. let's have a look at BrandGrid.qml, which creates a grid of the brand component.

```
import QtQuick 2.0
import QtQuick.Layouts 1.0
import QtQuick.Controls 2.0


Item {
    id: brandGrid
    signal productSelected(string name, string description, string price)

    function addToModel(item) {
        iceCreamSelection.append(item);
    }

    ListModel {
        id: iceCreamSelection
    }

    Component {
        id: iceCreamDelegate

        Brand {
            imageSrc: img
            product: productObject

            onSelected: {
                brandGrid.productSelected(name, description, price);
            }
        }
    }

    GridView{
        id: grid
        width: parent.width * 0.7
        height: parent.height
        anchors.horizontalCenter: parent.horizontalCenter
        cellWidth: 280
        cellHeight: 280
        model: iceCreamSelection
        delegate: iceCreamDelegate
        focus: true
        clip: true

        highlight: Rectangle { color: "lightsteelblue"; radius: 5; opacity: 0.5 }
    }
}
```

QML heavily uses the MVC (model-view-controller) paradigm. Data is stored in some model which is then rendered by a view that uses a "delegate" to render each item. In this example we have our ice cream products in a ListModel with the id "iceCreamSelection". This model is populated by calling the `addToModel(item)` function. As items are added to the model, the view will automatically render them without a further need to call any rendering function.

The view in this example is a GridView with the id "grid". It takes the model as a property value. Each item in the model is then rendered with a delegate called "iceCreamDelegate" which in actuality is the Brand component that was shown and analyzed in the previous chapter. Note how the products are set to Brand by the delegate and how the delegate captures the "selected" signal from the Brand component when the component is clicked. In this example the action to take after receiving the signal is to send yet another signal.

2.2.2. React version of component reuse

Likewise, do acheive the same BrandGrid in React, let's create a file BrandGrid.js. The BrandGrid component is shown in the snippet below:

```
import React, { Component } from 'react';
import Brand from './Brand.js'
import './BrandGrid.css';

class BrandGrid extends Component {

    constructor(props) {
        super(props);
        this.state = {}
        this.onBrandSelected = this.onBrandSelected.bind(this);
    }

    onBrandSelected(name, description, cost) {
        this.props.selectedCallback(name, description, cost);
    }

    render() {
        var cb = this.onBrandSelected;
        return (
            <div className="BrandGrid">
            { this.props.products.map(function(item, idx) {
                return (<Brand key={idx} product={item} selectedCallback={ cb } />)
            })}
            </div>
        );
    }
}

export default BrandGrid;
```

```
.BrandGrid {
 text-align: left;
 width: 900px;
 margin: 0px auto;
}
```

React does not nativaly have similar MVC components as QML's GridView but a grid is easy to draw and redraw as an HTML5 table or collection of divs. The stylesheets make these elements flow nicely and dynamically into a grid layout.

From the above snippets, we can deduce:

- Both QML and React allow the reuse of components and generating components dynamically.
- In QML, a component is linked to the model via the delegate property and the view will automatically multiply it according to the model. In React, the component can be used as a tag element which is then multiplied by a map or a loop to create the eventual rendering.
- Note that in QML if the code of a component resides in a file that is in the same directory as the file that uses it, no explicit imports are required to access the component. In React on the other hand each component must be added with an import statement.

3. Accessing Asema IoT objects

Finally, how do we load the data from the Asema IoT backend so that it can be rendered on a UI? Here, QML code has the advantage of running on the same software instance as the backend logic that automatically takes care of object data filtering and updates. Therefore the QML objects can load their data through a so called application context. With React, the data needs to be loaded using an AJAX call.

Now, although the procedure used to load the data is different, the actual code is exactly the same. This is because the context in QML is called using JavaScript coding that in turn makes an asynchronous call to the backend. For React, Asema IoT offers a set of libraries that feature exactly the same functions as the QML code and because they use AJAX, the calls are natively asynchronous.

So the procedures are the same, down to the final syntax.

3.1. Loading IoT objects with QML

3.1.1. Importing contexts

First, you need to import into your QML application the correct context. This is how to do it:

```
import Jarvis 1.0
```

The import command will supply you with a context called `objectManager`. By hooking this manager to a callback function and then calling one of the various search functions, the backend then loads the objects for you.

3.1.2. Loading objects

To load an object in QML, you need

1. To have a container context that stores the objects during runtime. The default context for this is the `screenletContext`
2. Hook the `objectsReady` signal of `screenletContext` to some callback method that processes it

Let's see how that is done:

```
function onLoadedObjectsReady(propertyName) {
 // Process the objects here
}

function open() {
 screenletContext.objectsReady.connect(onLoadedObjectsReady);
 // This assumes that all ice cream products have been tagged as "sample:IceCream"
 objectContext.findObjectsByMetaType(screenletContext, "products", "sample:IceCream");
}
```

3.2. Loading IoT objects with React

React applications can run as pure browser applications by importing the React libraries into an HTML page. With this method, all files originate from the same server, in this case the Aseme IoT Central HTTP server. However, a more common approach for building React applications is to use a node.js backend. In this case the application is created by using the npm tool "create-react-app" which provides the basic scaffolding to create a React app. However, this configuration now causes a problem: UI code is on one server (node.js) while the IoT objects are behind the JSON API of Asema IoT Central. Browsers don't usually like this setup as it is cross-domain. This means that a React app on its own can not access the Asema IoT backend to load and manipulate objects. To achieve that, we need to set up a common front for both servers to work around the problem. A potential common front is a proxy server, such as nginx.

3.2.1. Setting up nginx

Assuming your React app is running on port 3000 and the Asema IoT backend in port 8080, your nginx configuration file (nginx.conf) could look like this:

```
server {
        listen       80;
        server_name  localhost;

# Forward nodeJs HTTP Traffic
        location / {
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $remote_addr;
            proxy_set_header Host $host;
            proxy_buffering off;
            proxy_pass http://localhost:3000;
        }

 # Forward node.js websocket traffic
        location /sockjs-node{
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection "upgrade";
            proxy_pass http://localhost:3000;
        }

# Forward IoT Central HTTP traffic
        location /iot {
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $remote_addr;
            proxy_set_header Host $host;
            proxy_buffering off;
            proxy_pass http://localhost:8080;
        }

# Forward IoT Central websocket traffic
        location /ws {
            proxy_pass http://localhost:8081;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection "upgrade";
        }

# Forward IoT Central JSON API traffic
        location /json{
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $remote_addr;
            proxy_set_header Host $host;
            proxy_buffering off;
            proxy_pass http://localhost:8080;
        }
```

In the above configuration file, the nginx server at localhost:80 forwards the requests made to it to NodeJS server at localhost:3000. What does the magic however is the forwarding of endpoints /iot, /json to the Asema IoT server and the JSON API endpoint both running at Asema Iot Central server at localhost:8080 and /ws to the Websocket server at localhost:8081. The application itself will now run at localhost:80 so point your browser to that address to view it.

3.2.2. Loading objects

Once the nginx setup is complete, the IoT objects can now be accessed from the backend. However, we still need to import the required scripts. This can be done in the index.html file of the React application. In this example, your index.html file should have the following imports.

```
<script type="text/javascript" src="/iot/inc/libs/jquery.min.js"></script>
<script type="text/javascript" src="/iot/settings.js"></script>
<script type="text/javascript" src="/iot/inc/js/jarvis-1.0/network/json.js"></script>
<script type="text/javascript" src="/iot/inc/js/jarvis-1.0/network/websocket.js"></script>
```

```
<script type="text/javascript" src="/iot/inc/js/jarvis-1.0/objects/signal.js"></script>
<script type="text/javascript" src="/iot/inc/js/jarvis-1.0/objects/connectedobject.js"></script>
<script type="text/javascript" src="/iot/inc/js/jarvis-1.0/objects/objectmanager.js"></script>
<script type="text/javascript" src="/iot/inc/js/jarvis-1.0/objects/dataanalysismanager.js"></script>
<script type="text/javascript" src="/iot/inc/js/jarvis-1.0/objects/pluginmanager.js"></script>
```

Additionally, to be able to use the `objectManager` to load objects, you need to initiate one instance of it when the libraries have loaded:

```
<script type="text/javascript">
    var objectManager = null;
    function load() {
        objectManager = new ObjectManager();
    }
</script>
<script type="text/javascript">load();</script>
```

Remember to place the `load()` method to a place where it is invoked only after the libraries have been received.

After the required scripts are loaded, we can finally access the objects from Asema IoT Central server. It is a good idea to load the objects in the main file of the react app (App.js) so that the objects can be stored as props and propagated throughout the application. Likewise, the objects are loaded in the componentDidMount method of the App.js file to ensure that the React components are loaded and ready before accessing the IoT objects. For the sample application, the objects could be loaded as:

```
componentDidMount() {
    var onLoadedObjectsReady = (function(propertyName) {
        // Handle the objects here
    }).bind(this);

    window.objectManager.ready.connect(function() {
        window.objectManager.findObjectsByMetaType("products", "smartapi:IceCream");
    });
    window.objectManager.objectsReady.connect(onLoadedObjectsReady);
    window.objectManager.start();
}
```

The function above starts the `objectManager` and finds the respective objects based on their type. When the objects are ready and loaded, they can then be assigned to the state of the application and used to load data or manipulate IoT objects.