Asema Screenlet Developers Guide

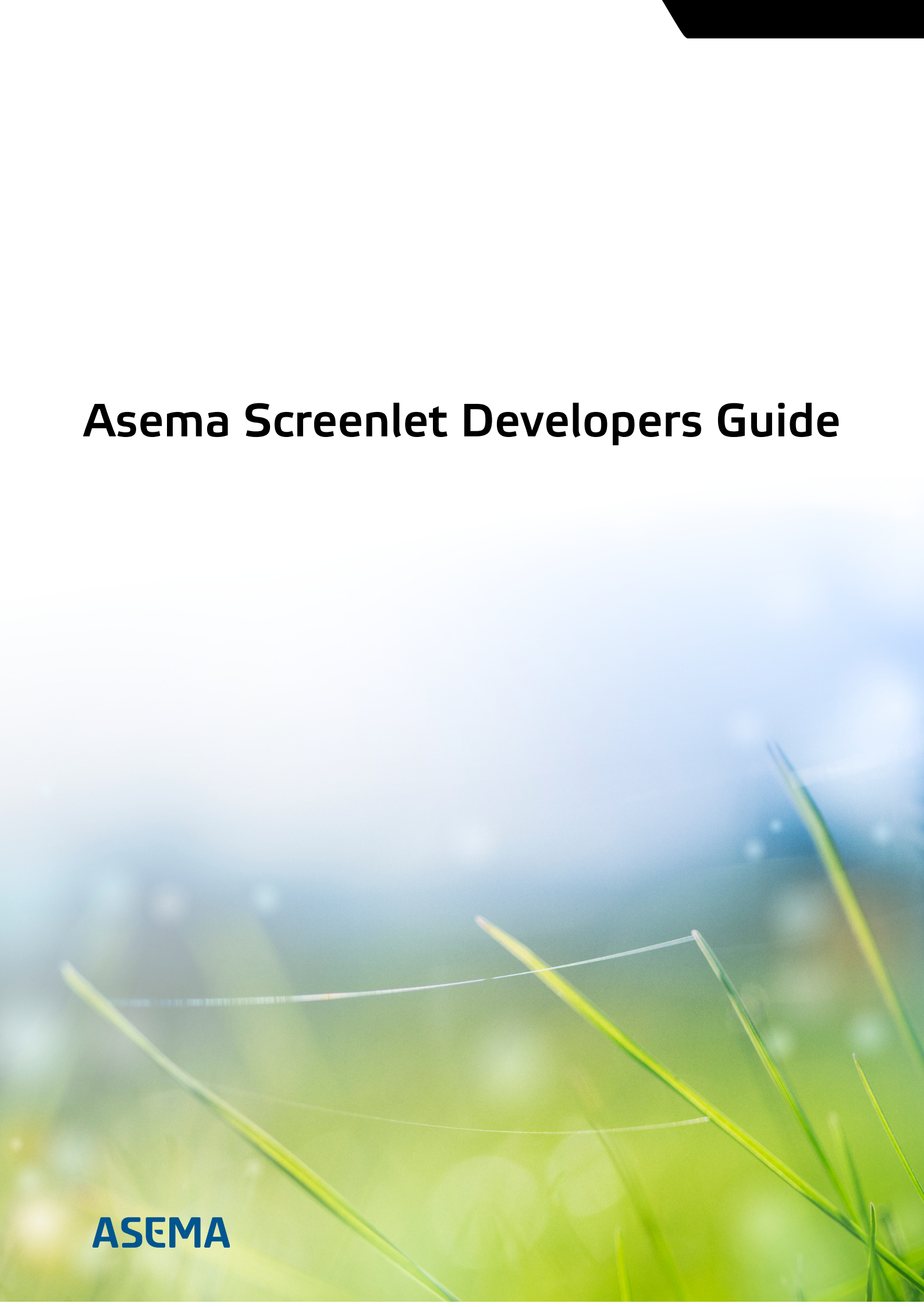

Table of Contents

1. Introduction

1.1. Introduction to Asema IoT Central

The Asema IoT Central is a versatile system integration software for Internet of Things projects. It is designed to be an efficient tool that scales without effort from simple one machine data logging applications to heavy-duty service oriented cloud architectures. Unlike the majority of IoT solutions that rely on a heavy middleware architecture, Asema IoT Central is one expandable modular application. The lightweight approach enables running it much more efficiently, with less hardware requirements, easier maintenance, and native capability for edge computing.

Asema IoT Central focuses on speed, efficiency and precision. Likewise, it is also designed to work across multiple platforms and architecture. Asema IoT offers three different programmable user interface methods; screenlets, HTML5 applications and reports. Each method has its own purpose, but all the methods follow a similar logic and to a large extent similar syntax so that engineers can flexibly choose the most appropriate approach for given application and use case.

1.2. Introduction to Screenlets

Screenlets are widgets or lightweight applications that can be placed on Asema IoT dashboard. They are familiar to programming concepts such as “applets” or “widgets” as each of these rely on underlying operating system to offer standardized methods for development, installation, configuration and removal of the program. The objective of screenlets is to offer developers easier and faster mean to develop desktop and mobile applications for Asema IoT. They can be used to add or augment functionality, create services or just to change the look and feel of the device software. Screenlets are meant to be straightforward, lightweight graphical applications that use underlying interfaces to perform complex operations. However, screenlets can also have multiple screens, menus, user interaction and significant amount of application logic and other features that may be found in a full-blown application software.

Screenlets are a mixture of markup and JavaScript, so most user interface programmers will feel right at home when developing screenlets. The declarative nature of the markup, however, gives a native and much granular set of tools compared to traditional HTML/CSS. Likewise, screenlets are not limited in ways JavaScript apps are on web browsers because as opposed to web apps which run on browser, Screenlets run on device. As they run on device, screenlets also offer better integration possibilities into the underlying hardware and software. Likewise, since screenlets run on any platform where Asema IoT can be installed therefore the screenlets you develop can also be used with Asema IoT Dashboard application on your mobile devices. This saves you the trouble of developing native applications for multiple platforms.

2. Getting Started

This section focuses on setting up your system for developing screenlets for Asema IoT. We assume you have already installed the required applications in your system.

2.1. Developer Mode

The developer mode adds additional functionality to the Asema E simulator that comes bundled with the Asema IoT Central application.The developer mode allows you to reload your simulator so that you don't have to restart it each time you make changes to your screenlet. LIkewise, developer mode offers a screenlet management interface to create and manage screenlets. The screenlet management interface also offers the ability to install screenlet packages and also to package or publish the application. In addition to that, developer mode allows scaling of the application to various sizes so that it is easier to see how the application behaves in different sizes. However, the most important feature of developer mode is logging as it shows you the system log and helps you debug your system as well as your application.

To access the developer mode, simply add a **-D** flag when launching the app. For example, in windows, you can navigate to the installation directory of Asema IoT Central and launch the application with a -D flag. I.e

```
Asema_IoT_Central_for_Windows.exe -D
```

Similarly, in Linux, from the installation directory of Asema_IoT_Central

```
./asema_iot_central.sh -D
```

2.2. Getting started with QML

QML (Qt Markup Language / Qt Markup Language) was developed by the Qt Project.

2.3. Creating a Screenlet

There are several ways to create a screenlet. The methods are discussed below

- With the Asema IoT Central admin interface

  1. Go to application management in your Asema IoT Central admin interface (`http://local-host:8080/p/applications`)
  2. Navigate to **Screenlet apps>Local Store**
  3. Now you can see a form where you can enter the name for the screenlet. A new blank screenlet is created when you submit the form with the create button,

- With Asema E simulator

  1. Click on the screenlet management icon in the Asema E interface (second icon from the right)
  2. In the manage tab of the pop up window, you can see a similar form where you can enter the name of the screenlet. Clicking on Create new button creates a new blank screenlet with the given name.

The screenlets created with the admin interface of with the Asema E simulator are available in Linux systems at

```
/home/<user>/.local/share/Asema IoT Central for Linux/screenlet_installs/local
```

and in Windows systems at

`\Users\<user>\AppData\local\Asema IoT Central for Windows\screenlet_installs\local`.

Alternatively, you can also create new screenlets by creating subdirectories inside the local screenlet directories mentioned above.

2.4. Your first screenlet

As it is customary in our industry, we'll start off with a good old "Hello World!".The basic screenlet in this example is a single file, called Screenlet.qml. You can either create this with one of the screenlet creation methods mentioned in section 3 or simply create inside the screenlet directory. The screenlet example presented below displays the classic "**Hello World!**" text in the center of the screen.

```
import QtQuick 2.3

Item {
    anchors.fill: parent
    function open() {}
    function close() {}

    Text {
         id: helloW;
         text: "Hello World!";
         anchors.centerIn: parent;
         color: "white";
         font.pointSize: 12;
          }
}
```

In the above example, you may have noticed that we imported `QtQuick`, something we have not mentioned before. QtQuick is the standard library for writing QML applications. IT provides the visual canvas and and includes tools for creating functional applications with QML.

As it is just a hello world application, most of the stuff happening there is pretty simple. We just created a item that fills the parent display and simply displays the text "**Hello World!**" in the middle of the screen. The functions `open()` and `close()`will be put to use and described in more detail in the following chapters.

3. Developing Screenlets

In this section. we will walk through screenlets on progressively higher complexity.

3.1. Simple Single Page Screenlets

3.1.1. Calculator

In this example we will be designing and implementing one of the simplest yet most useful tools, a calculator. A normal calculator has functions to perform elementary mathematical calculations like addition, subtraction, multiple and division. The calculator application will have two components; the interface, written in QML and the calculator logic, written in JavaScript. First, let's have a look at the calculator UI.

3.1.1.1. Calculator UI

A calculator has buttons to represent numbers and operations it can perform. Those buttons are usually arranged in a grid below the display that displays keystrokes, operations and the results of those mathematical operations. Since we are going to need multiple buttons in our UI, it is a wise to create a template for the buttons and use those in our main UI.

3.1.1.2. Buttons

A calculator button is rectangular, is labeled with the operation or number it represents and has the ability to be clicked or toggled. Therefore, the buttons in our UI should be able to send a clicked signal, when a button is pressed. The file we are creating is called `CalcButton.qml` and it is created inside the `src` directory. The code snippet below shows a button design:

```
import QtQuick 2.3

Item {
    property string operation
    property string label
    property bool togglable: false
    property bool toggled: false
    signal clicked

    id: button; width: 50; height: 30

    BorderImage{
        id: button_bg
        source: "../img/button.sci"
        anchors.fill: parent
    }

    //Properties of the button text
    Text {
        id: labelText;
        text: label.length ? label : operation
        anchors.centerIn: parent;
        color: "white"
        font.pixelSize: 16
        font.bold: true
    }

    //Defining mouse area to catch click event
    MouseArea {
        id: clickRegion
        anchors.fill: parent
    }

    states: [
        State {
            name: "Pressed"; when: clickRegion.pressed == true
            PropertyChanges { target: button_bg; source: "../img/button_pressed.sci" }

        },
        State {
            name: "Toggled"; when: button.toggled == true
            PropertyChanges { target: button_bg; source: "../img/button_pressed.sci" }
        }
    ]
}
```

In the above example, we defined a template for a button. The button we defined is a rectangle of 50 X 30 units, has a text label and can be clicked to change its state to clicked or toggled.

3.1.1.3. Designing the Calculator UI

To create out calculator UI, we are going to import the button templates and arrange the created buttons in a grid and set their properties. The application UI is designed using a grid structure. The diagrammatical representation of the grid is provided below:

Likewise, you can find the QML designs in the `screenlet.qml` sample snippet below

```
import QtQuick 2.3
import Jarvis 1.0
import "src" as CALC //to import the buttons template we designed earlier

//Calculator screenlet

Item {
    id: frame
    anchors.fill: parent

    width: deviceContext.screenWidth
    height: deviceContext.screenHeight

    property ScreenletContext screenletContextLike

    state: (deviceContext.orientation == DeviceContext.Portrait) ? 'PRT' : 'LSC'


    property int numButtonWidth: 50
    property int numButtonHeight: 34
    property int funcButtonWidth: 50
    property int funcButtonHeight: 34
    property int cButtonWidth: 60
    property int cButtonHeight: 34
    property int mButtonWidth: 40
    property int mButtonHeight: 34
    property int buttonSpacing: 6

    function open() {
    }

    function close() {
    }

    Grid {
        id: frameGrid

        rows: 2
        spacing: 6

        x: 125
        y: 0


        // Display area for results and operations
        Rectangle {
            id: displayRect
            color: "#eeeeaa"
            radius: 5
            width: parent.width
            height: 45

            Text {
                id: curNum
                font.bold: true
                font.pixelSize: 16
                anchors.verticalCenter: parent.verticalCenter
                anchors.right: parent.right
                anchors.rightMargin: 5

            }

            Text {
                id: currentOperation
                font.bold: true; font.pixelSize: 16
                anchors.left: parent.left
                anchors.leftMargin: 5
                anchors.verticalCenter: parent.verticalCenter
            }

        }

        Grid{
            id: buttonGrid
```

```
            rows: 2
            columns: 2
            spacing: buttonSpacing

            Grid {
                id: numButtonsGrid
                columns: 3
                spacing: buttonSpacing

                CALC.CalcButton {
                    width: numButtonWidth
                    height: numButtonHeight
                    operation: "7"
                }

                CALC.CalcButton {
                    width: numButtonWidth
                    height: numButtonHeight
                    operation: "8"
                }

                CALC.CalcButton {
                    width: numButtonWidth
                    height: numButtonHeight
                    operation: "9"
                }

                CALC.CalcButton {
                    width: numButtonWidth
                    height: numButtonHeight
                    operation: "4"
                }

                CALC.CalcButton {
                    width: numButtonWidth
                    height: numButtonHeight
                    operation: "5"
                }

                CALC.CalcButton {
                    width: numButtonWidth
                    height: numButtonHeight
                    operation: "6"
                }

                CALC.CalcButton {
                    width: numButtonWidth
                    height: numButtonHeight
                    operation: "1"
                }

                CALC.CalcButton {
                    width: numButtonWidth
                    height: numButtonHeight
                    operation: "2"
                }

                CALC.CalcButton {
                    width: numButtonWidth
                    height: numButtonHeight
                    operation: "3"
                }

                CALC.CalcButton {
                    width: numButtonWidth
                    height: numButtonHeight
                    operation: "0"
                }

                CALC.CalcButton {
                    width: numButtonWidth
                    height: numButtonHeight
                    operation: "."
                }

                CALC.CalcButton {
                    width: numButtonWidth
                    height: numButtonHeight
                    operation: "+/-"
                    label: "±"
                }
            }

            Grid {
                id: funcButtonsGrid
                rows: 4
                spacing: buttonSpacing

                CALC.CalcButton {
                    width: funcButtonWidth
                    height: funcButtonHeight
                    operation: "+"
                }

                CALC.CalcButton {
                    width: funcButtonWidth
```

```
                    height: funcButtonHeight
                    operation: "-"
                }

                CALC.CalcButton {
                    width: funcButtonWidth
                    height: funcButtonHeight
                    operation: "x"
                }

                CALC.CalcButton {
                    width: funcButtonWidth
                    height: funcButtonHeight
                    operation: "/"
                }
            }

            Grid {
                id: cButtonsGrid
                columns: 2
                spacing: buttonSpacing

                CALC.CalcButton {
                    width: funcButtonWidth
                    height: funcButtonHeight
                    operation: "Bksp"
                    label: "<"

                }

                CALC.CalcButton {
                    width: funcButtonWidth
                    height: funcButtonHeight
                    operation: "C"
                }

            }

            CALC.CalcButton {
                width: funcButtonWidth
                height: funcButtonHeight
                operation: "="
            }
        }
    }

    states: [
        State {
            name: "PRT"

        PropertyChanges {
                target: frameGrid
                x: 30
                y: 40
            }

            PropertyChanges {
                target: frame
                mButtonHeight: 60
            }
        }
    ]
}
```

Now that our calculator UI is ready, but it can't perform any calculations yet. Therefore, it is now time to implement actual calculation logic. The calculator logic, as most application logic in Asema IoT applications is defined in a JavaScript file. For our purpose, we will create a file `calculator.js` inside the `src/` directory of the application (That's the place where the buttons template is).

3.1.1.4. Programming application logic

As mentioned earlier, the application logic is defined in a JavaScript file. The JavaScript files containing application logic are placed inside the `src` directory. The file we created is named `calculator.js` Since we are designing a calculator and JavaScript has built in functions for most calculations, this should not be a tough task but here's a snippet anyway.

```
var curVal = 0;
var memory = 0;
var lastOp = "";
var timer = 0;

//checking disabled operations
function disabled(op) {
    if (op == "." && curNum.text.toString().search(/\./) != -1) {
        return true;
```

```
    } else if (op == "Sqrt" &&  curNum.text.toString().search(/-/) != -1) {
        return true;
    } else {
        return false;
    }
}

//handling operations
function doOp(op) {
    if (disabled(op)) {
        return;
    }

    //handling numbers
    if (op.toString().length==1 && ((op >= "0" && op <= "9") || op==".") ) {
        if (curNum.text.toString().length >= 14)
            return; // No arbitrary length numbers
        if (lastOp.toString().length == 1 && ((lastOp >= "0" && lastOp <= "9") ||                                    lastOp==".") )
 {
            curNum.text = curNum.text + op.toString();
        } else {
            curNum.text = op;
        }
        lastOp = op;
        return;
    }
    lastOp = op;

    // Pending operations
    if (currentOperation.text == "+") {
        curNum.text = Number(curNum.text.valueOf()) + Number(curVal.valueOf());
    } else if (currentOperation.text == "-") {
        curNum.text = Number(curVal) - Number(curNum.text.valueOf());
    } else if (currentOperation.text == "x") {
        curNum.text = Number(curVal) * Number(curNum.text.valueOf());
    } else if (currentOperation.text == "/") {
        curNum.text = Number(Number(curVal) / Number(curNum.text.valueOf())).toString();
    } else if (currentOperation.text == "=") {
    }

    if (op == "+" || op == "-" || op == "x" || op == "/") {
        currentOperation.text = op;
        curVal = curNum.text.valueOf();
        return;
    }
    curVal = 0;
    currentOperation.text = "";

     // plus/minus
    if (op == "+/-") {
        curNum.text = (curNum.text.valueOf() * -1).toString();
    }

    //backspace
    else if (op == "Bksp") {
        curNum.text = curNum.text.toString().slice(0, -1);
    }

    //Clear display
    else if (op == "C") {
        curNum.text = "0";
    }
}
```

In the above snippet we defined the application logic for our calculator. Once you create and save the file, you can try to reload the screenlet. You'll find that your calculator still does not work, that is because we still have couple more things to do.

3.1.1.5. Importing and adding application logic

This is the final thing to do. We have both out user interface and our application logic ready, all we have to do is to import and integrate our application logic to the user interface. To import a file, you can simply use the import keyword of QML. You have to import your JavaScript file in the main `screenlet.qml` file to access the application logic it holds. You can add the snippet below to import your .js file.

```
import "src/calculator.js" as CALCEngine
```

Importing the logic is not the only thing to do. We still have to use the functions to perform our calculations. As we know, each button in a calculator performs a different operation when it is pressed, we will add the `doOp()` function in the button template itself. However, you also have to add a reference in the main screenlet body. The reference is added inside the main frame definition in screenlet.qml file.

```
function doOp(c) {CALCEngine.doOp(c)}
```

In the snippet above, the function `doOP(c)` executes the function `doOp(c)` where c is the desired operation as defined in the calculator.js file. Finally, we also have to link each button with the calculator engine. To perform that, we call the `doOp(operation)` function when the click event is triggered. The snippet below displays the code for the onClicked event of the calculator button in `CalcButton.qml.`

```
onClicked: {
            doOp(operation);
            button.clicked();
            if (!button.togglable) return;
            button.toggled ? button.toggled = false : button.toggled = true
        }
```

Once you have done all that, you should now have a functional calculator screenlet.

3.1.2. Reminder

Reminders are one of the most commonly used tools,because you know, we forget things. So we set reminders for everything from buying milk to going to the bank. Fore reminder or TODO applications, we require a database to write our new tasks to and mark tasks done. Likewise, we also require an user interface that lets the users view and edit tasks and mark the tasks done in order to remove the task from the TODO list. In this example, we will develop a simple reminder screenlet that displays the list of existing tasks and lets the users organise those tasks accordingly.

During this development, we will also be using some new elements from the QT framework like `List-Model` and `ListView` and we will also introduce the concept of delegates.

3.1.2.1. Reminder UI

Our application UI will be pretty simple. It is a `ListView` containing the name of the task along with two buttons; one to mark tasks done and another to delete the task entirely. Similarly, we'll also be making use of `Component` in QML to define a delegate to arrange the items from the Asema IoT inbox in out `ListView`.

```
import QtQuick 2.3
import Jarvis 1.0


// Reminder screenlet
Item {
    id: screen
    width: deviceContext.screenWidth
    height: deviceContext.screenHeight

property ScreenletContext screenletContext

    state: (deviceContext.orientation == DeviceContext.Portrait) ? 'PRT' : 'LSC'

    //Defining the delegate to display task list
    Component{
        id: taskDelegate

        Rectangle {
        id: taskRect
            height: 63
            width: parent.width
            border.color: "#95C7E0"
            border.width: 1

            color: model.index % 2 == 0 ? "#336C97" : "#3979A9"
            radius: 8

           Row {
           spacing: 2

           Item {
           height: parent.height
           width: taskRect.width - 2*63

               Text {
               anchors.fill: parent
```

```
               anchors.topMargin: 5
               anchors.bottomMargin: 5
               anchors.leftMargin: 10

            text: modelData.data['task']
            font.pixelSize: 20
            font.family: "Elektra Text Pro"
            color: modelData.data['done'] ? "#292929" : "white"
            font.strikeout: modelData.data['done']
            verticalAlignment: Text.AlignVCenter
            wrapMode: Text.WordWrap
            clip: true
        }
        }

           Image {
           id: doneButton
           height: 63
           width: 63
            source: "img/button_normal.png"

            Image {
            source: "img/icon_check.png"
            anchors.verticalCenter: parent.verticalCenter
        anchors.horizontalCenter: parent.horizontalCenter
        fillMode: "PreserveAspectFit"
            }

            MouseArea {
            anchors.fill: parent


            }
        }

        Image {
        id: removeButton
        height: 63
           width: 63
            source: "img/button_normal.png"

            Image {
            source: "img/icon_delete.png"
            anchors.verticalCenter: parent.verticalCenter
        anchors.horizontalCenter: parent.horizontalCenter
        fillMode: "PreserveAspectFit"
            }

            MouseArea {
            anchors.fill: parent

            }
        }
    }
        }
    }


    ListView {
        id: task_list

        model: screenletContext.inbox
        delegate: taskDelegate

        anchors.fill: parent
anchors.bottomMargin: 42
        anchors.topMargin: 20
        anchors.leftMargin: 20
        anchors.rightMargin: 10

        interactive: true
        clip: true
        snapMode: ListView.SnapToItem
    }

Text {
id: entries
width: parent.width
height: 20
font.pixelSize: 15
font.family: "Elektra Text Pro"
color: "white"
anchors.bottom: parent.bottom
anchors.bottomMargin: 8
horizontalAlignment: Text.AlignHCenter
}

    Connections {
        target: systemContext
    }

    //Checking for data and displaying message if null
    function open() {
        screenletContext.numPages = 0;
        if (screenletContext.inboxSize == 0)
        error_message.opacity = 1;
```

```
    }


    //this error message is shown if inbox is empty
    Rectangle {
        id: error_message
        opacity: 0
        anchors.fill: parent
        anchors.bottomMargin: 48
        anchors.topMargin: 30
        anchors.leftMargin: 20
        anchors.rightMargin: 20
        radius: 10
        border.color: "white"
        border.width: 1

        color: "#3979A9"

        Behavior on opacity {
            NumberAnimation { property: "opacity"; duration: 1000 }
        }

        //Error message for empty Inbox
        Text {
            id: error_text
            width: deviceContext.screenWidth - 100
            wrapMode: Text.WordWrap
color: "#ffffff"
            font.family: "Elektra Text Pro"
            font.pixelSize: 18
            anchors.horizontalCenter: error_message.horizontalCenter
            anchors.verticalCenter: error_message.verticalCenter
            text: qsTr("Your list of things to remember is empty.")
        }

//Getting task data from the inbox

    Connections {
        target: screenletContext
        onInboxSizeChanged: {
        if (size == 0 && error_message.opacity == 0) {
        error_message.opacity = 1;
        } else if (size > 0 && error_message.opacity == 1) {
        error_message.opacity = 0;
        }
        }
    }
    }
}
```

In the above snippet, we designed a ListView interface to display the list of tasks from the Asema IoT Inbox. Initially, we defined a `Component` to delegate how the tasks we fetch are displayed in the `ListView`. The `ListView` element can take a **delegate** and model to display the model data as defined in the delegate component. The delegate we have defined adds two buttons, one for removing the task and the other for marking it done to every element in the data model. Once we have the UI ready, we can now add functionality to the buttons and handle those events to make changes to the tasks.

3.1.2.2. Handling Events

As we are all familiar with reminder applications, the possible events that we may face are the user clicking on remove task or mark the task done menu. First, when the user presses the done button, the task is supposed to be marked done and then removed from the database. The code snippet below is an example event handler or the done button. This code goes in the `MouseArea` definition of the done button.

```
onReleased: {
             doneButton.source = "img/button_normal.png"
            }

            onPressed: {
             doneButton.source = "img/button_pressed.png"
            }

         onClicked: {
            var d = modelData.data;
            d['done'] = !d['done']
            modelData.setData(d);
            recount();
            }
```

In the above snippet, once the done button is pressed, we triggered a recount. The method for recount is provided later in this section. But before that, we have another button to attend to. Below is the sample event handler for the **Remove** button. Similar to the Done button, the event handlers are defined in a `MouseArea` to catch mouse click events and process them.

```
onReleased: {
    removeButton.source = "img/button_normal.png"
}
onPressed: {
    removeButton.source = "img/button_pressed.png"
}

onClicked: {
    screenletContext.removeInboxItem(model.index);
}
```

The snippet above removes the inbox item corresponding to the selected task to remove.

3.1.2.3. Updating data

We have a user interface ready and we've also set up buttons. All we need to do now is to actually update the database and trigger a recount to update the UI each time a task is marked done or deleted. First things first, don't forget to update your data once your component (delegate) completes doing its job. To acheive that, simply add the following snippet to your code, just after the component definition.

```
Component.onCompleted: {
    modelData.append //appending the data
}
```

Once we update the data, we also have to trigger a recount of the tasks to take into account the tasks in the list that are marked done or are removed. The example reload function definition is provided in the snippet below

```
function recount() {
    var done = 0;
    for (var i = 0; i < screenletContext.inboxSize; i++) {
        var data = screenletContext.getInboxItemData(i);
        if (data['done']) done++;
    }
    entries.text = qsTr("Tasks done: ") + done + "/" + screenletContext.inboxSize
}
```

The `recount()` function above recounts the tasks when a task is marked done and displays the number of tasks completed. The `reload()` function defined above is called whenever a reload is needed; e.g. When the inbox size is changed (`onInboxSizeChanged` event of the screenet context, or the open function for the initial count.). Once the events are handled properly and the tasks recounted and views reloaded when necessaary, your reminder app will be ready. You can now make reminders about everything from buying potatoes to repairing your Mr.Potato toy.

4. Managing Screenlets

5. Publishing Screenlets