Book 2 in a series of 3

Lomiri App Development Level 2

Brave New Books

Lomiri App Development Level 2

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.

Publisher
First printed in March 2026 by Brave New Books
ISBN: 9789465389257

When freedom is no longer a choice, you wish you had acted differently when you had the opportunity. It’s not only our job, but also yours, to make sure that our children and grandchildren, still will have a freedom of choice. Our software is ours, Our data is ours, the only way to guarentee our freedom is by using, stimulating and contributing to free and open source.

– Ir. Erik Mols

Preface

Welcome to the second book in a series of three. In the first book we started with developing Lomiri apps with Clickable on the Ubuntu Touch platform. We ended with a simple app. In this book we delve on forwards and discuss much more of Lomiri apps. We will emphasize on Lomiri on Ubuntu touch, but also will show, how you can develop SNAP apps for Lomiri on other Linux operating systems, like for Lomiri on Debian, Arch or Rhino. Lomiri is the future as convergence will be the standard in the world. Imagine how you get out of bed in the morning, check your mail on your smartphone, then after breakfast when you start to work, you click your phone in a docking-station and it turns on a full fledged working environment on a big screen, with mouse and keyboard. Imagine that all the apps then run native on your desktop. Is this the future??? This is reality, as Lomiri is already capable of this.

This book is written by OS-SCi.

Video content

Where the original course contained videos, this book contains QR-codes, which point to the video's.

1 Introduction

1.1 Ubuntu Touch App Development – Level 2

This book builds upon the foundation established in the first book, taking your skills to the next level with a focus on advanced UI design, interactivity, and convergence support. While Level 1 introduced you to QML, JavaScript, and Clickable for building and publishing apps, Level 2 explores how to craft *rich, dynamic, and responsive* user experiences on the Ubuntu Touch platform.

This book emphasizes the creation of professional-grade applications that make full use of the Lomiri design system. You'll learn to integrate features like multi-pane layouts, swipe gestures, floating action buttons, and collapsible views that enhance usability across both mobile and desktop environments.

Developed in collaboration with the UBports community, this course continues OS-SCi's mission to empower developers with open technologies. Each lesson builds practical experience, blending hands-on projects with conceptual understanding to prepare you for real-world Ubuntu Touch development. In this book we not only discuss the technologies, but we also give a thorough example of an app communicating with the backend, this is done in chapter 18. Where we talking mainly in the first book and this book about developing apps with clickable, we are going to look in to SNAP apps, which also run on Ubuntu, Debian, Arch and Rhino Linux desktops. This is done in the packaging chapter 21.

1.2 Course Outline

Chapter 1 Leading and Trailing Actions

Chapter 2: Implementing Swipe Gestures

Chapter 3: Designing Multi-Pane Views for Convergence

Chapter 4: Floating Action Buttons (FABs) and Menus

Chapter 5: Option and Combo Selectors for Dynamic Input

Chapter 6: Radio Buttons and Multi-Option Selectors

Chapter 7: Expansion Options

Chapter 8: Tabbed Views, Search Features, and Info Dialogs, Repeaters

Chapter 9: Date Range Selectors, Progress Bars, and Collapsible Views, Hierarchical List View

Chapter 10: Theme Preferences, Dark Mode, and Content Hub Integration

Chapter 11: Rich Text Editor

Chapter 12: Splash Screen

Chapter 13: Charts

Chapter 14: Model View Architecture

Chapter 15: Favroite Feature

Chapter 16: Background Timer

Chapter 17: Weblate

Chapter 18: Sync With Odoo Server

Chapter 19: Partial Sync

Chapter 20: Push Notification

Chapter 21: Packaging

Ubuntu Touch Level 2 continues to use QML for user interfaces and JavaScript or Python for logic and backend integration. You'll gain practical insight into responsive UI design, user input handling, and media attachment support, while also learning how to deliver apps that adapt gracefully across different form factors through Ubuntu's convergence principle.

By the end of this course, you will be able to design, build, and deploy advanced Ubuntu Touch applications that integrate modern mobile features, deliver fluid user experiences, and maintain Ubuntu's core values of openness, privacy, and innovation.

2 Study Goals

By the end of this course, you will be able to:

- Implement leading and trailing actions to create responsive list item behaviors and contextual interactions.
- Integrate swipe gestures (e.g., swipe up to create a timesheet) to enhance user experience and streamline task execution.
- Design multi-pane layouts that support Ubuntu Touch's convergence feature, enabling seamless transitions between mobile and desktop modes.
- Utilize floating action buttons (FABs) and FAB menus for quick access to essential actions such as creating timesheets, tasks, or activities.
- Configure and apply option selectors, combo selectors, and radio buttons for flexible and intuitive data selection.
- Develop expansion options for description fields and incorporate rich text editors to support detailed user input and formatting.
- Build tabbed views to organize content under multiple filters, combined with search features for efficient data retrieval.
- Implement info dialog boxes, progress bars, and date selectors (including date range pickers) to enhance interactivity and feedback.
- Create multi-option selectors (e.g., multi-assignee selectors), repeaters, and hierarchical list views with parent-child structures for complex data management.
- Apply collapsible views to display or hide content dynamically, improving readability and layout efficiency.
- Enable theme preference settings and dark mode support to enhance accessibility and user customization.
- Integrate content hub functionality for managing attachments and providing media support—allowing users to view and download images, PDFs, and documents.

3 How the Ubuntu Touch OS is different from Other Mobile OS

3.1 Why Ubuntu Touch? Key Differences and Advantages

Why Ubuntu Touch? Key Differences and Advantages

Ubuntu Touch is a Linux-based mobile operating system designed with a strong focus on privacy, openness, and user control. While Android and iOS dominate the mobile market, Ubuntu Touch provides a unique alternative for users and developers who prefer an open-source ecosystem and a more transparent platform.

3.2 Key Differences

Unlike Android and iOS, Ubuntu Touch is built on open-source foundations and follows a Linux-style development approach. This gives developers more flexibility to understand how the system works, customize applications, and contribute directly to the platform and its community.

Ubuntu Touch also supports convergence, meaning the same application can adapt its interface across different form factors such as phones, tablets, and desktops. This makes it possible to build apps that scale smoothly across screen sizes without maintaining separate versions for each device type.

Advantages of Ubuntu Touch
• Open-source platform: Developers can inspect, modify, and contribute to the operating system and

application ecosystem.
• Privacy-focused: Ubuntu Touch is designed to minimize unnecessary data collection and gives users more control over permissions and access.
• Convergence support: Applications can be designed to work across multiple screen sizes and device types.
• Community-driven ecosystem: Development is supported by an active open-source community, encouraging collaboration and innovation.

In the following lessons, we will explore how Ubuntu Touch application development differs from traditional mobile platforms and how to implement key UI features using Lomiri and Ubuntu Touch frameworks.

Contents

Leading and Trailing Actions 1

1.1 Introduction

In modern user interfaces, efficiency and intuitiveness are key components of a positive user experience. One popular interaction pattern that enhances usability in list-based views is the use of swipe actions. These allow users to quickly access common operations like editing, deleting, or marking items without navigating through multiple menus or leaving the current screen.

Leading and trailing actions refer to the interactive options revealed when a user swipes a list item either from left to right or right to left. Leading actions are typically triggered by a swipe from the left (trailing the finger to the right), while trailing actions appear during a swipe from the right. This technique saves valuable screen space and streamlines workflows by making important tasks accessible with a simple, fluid gesture.

In this lesson, you will learn how to implement leading and trailing swipe actions in an Ubuntu Touch application, define custom buttons for specific gestures, and connect these actions to your app logic to modify list data dynamically.

1.2 Leading and Trailing Actions

In this lesson, you will learn how to implement and customize `leading` and trailing swipe actions using the `ListItem` component. We will explore the key properties, behaviors, and customization options that help create a smooth and responsive user experience.

1.2.1 Terminology

To understand leading and trailing actions, it's important to get familiar with a few basic terms:

- *ListItem:* This is a single row or element in a list, like one email in your inbox or one contact in your phonebook.
- *Leading and Trailing:* These terms describe the sides of the list item based on the reading direction of the language you use:
- *Leading* means the *start side* (usually the left side in languages that read left-to-right, like English).
- *Trailing* means the *end side* (usually the right side in left-to-right languages).
- *Swipe Gesture:* This is the action of dragging your finger across the screen. In this context, swiping left or right on a list item reveals special actions.

- *LTR and RTL:*
  - *LTR (Left-To-Right)* is when text and interfaces flow from left to right, like English or Spanish.
  - *RTL (Right-To-Left)* is when they flow from right to left, like Arabic or Hebrew. The leading and trailing sides switch accordingly.
- *Action:* A task or command the user can trigger, such as "Edit," "Delete," or "Share," usually represented by an icon and a label.
- *Property:* A setting or characteristic of a component that controls its behavior or appearance, like whether swiping is enabled or what actions are shown.

`ListItem` - It is the main component, Provides `leadingActions`, `trailingActions` and `swipeEnabled` gesture behavior.

1.2.2 Explanation

- `leadingActions` — The `leadingActions` property of the `ListItem` component defines a set of actions that are revealed when the user swipes the item from left to right (in LTR locales). These actions appear on the leading side of the list item.
- `trailingActions` — The `trailingActions` property defines a group of swipeable actions that appear on the trailing side of a `ListItem`, when the user swipes from right to left (in left-to-right locales).
- `swipeEnabled` — The `swipeEnabled` property controls whether a `ListItem` allows swipe gestures to reveal its `leadingActions` and `trailingActions` panels.
- `ListItemActions` — Used to define multiple `Action` components for either leading or trailing side.
- `Action` — Defines the actual operation (with icon, label, signal).

Code Example:

```
import QtQuick 2.4
import Lomiri.Components 1.3
Item {
    anchors.fill: parent
    Column {
        anchors.fill: parent
        spacing: units.gu(2)
        anchors.margins: units.gu(2)
        ListView {
            width: parent.width
            height: parent.height - units.gu(10)
            clip: true
            model: ListModel {
                ListElement { title: "Item 1"; subtitle: "With leading
     icon" }
            }
            delegate: ListItem {
                width: parent.width
                height: units.gu(8)
                leadingActions: ListItemActions {
                    actions: [
                        Action {
                            iconName: "delete"
                            onTriggered: console.log("Delete", index)
                        }
                    ]
                }
                trailingActions: ListItemActions {
                    actions: [
                        Action {
                            iconName: "edit"
                            onTriggered: console.log("Edit", index)
                        }
                    ]
                }
                Row {
                    anchors.left: parent.left
                    anchors.leftMargin: units.gu(2)
                    anchors.verticalCenter: parent.verticalCenter
                    spacing: units.gu(1.5)
                    Column {
                        anchors.verticalCenter: parent.verticalCenter
                        Label {
                            text: title
                            font.pixelSize: units.gu(2)
                        }
                        Label {
                            text: subtitle
                            font.pixelSize: units.gu(1.5)
                            color: LomiriColors.darkGrey
                        }
                    }
                }
            }
        }
    }
}
```

1.2.3 Action Properties

- iconName - Specifies the name of the icon to display for an Action within a ListItem, typically referencing a predefined icon from

the app's icon set.
- text - Defines the label or caption displayed alongside the icon in an Action, describing the action's purpose (e.g., "Edit", "Delete").
- onTriggered - Specifies the callback function to execute when the Action is activated (e.g., tapped or clicked by the user).
- enabled - A boolean property that determines whether the Action is interactive; if set to false, the action appears disabled and cannot be triggered.
- visible - A boolean property that controls the visibility of the Action; when set to false, the action is hidden and not shown in the UI.

ListItemStyle - Customizes how the leading/trailing panels are shown (animations, clipping, etc.).

- animatePanels - A boolean property that enables or disables animation effects when showing or hiding the leading and trailing action panels during swipe gestures.
- snapAnimation - Defines the transition animation used to smoothly snap the swipe panels open or closed after a swipe gesture completes.
- swipeEvent() - A handler function or signal that responds to swipe gesture status updates, such as Started, Updated, or Finished, allowing custom behavior during swipe interactions.

SlotsLayout - Manages placement of components inside list items (icon, switch, etc.).

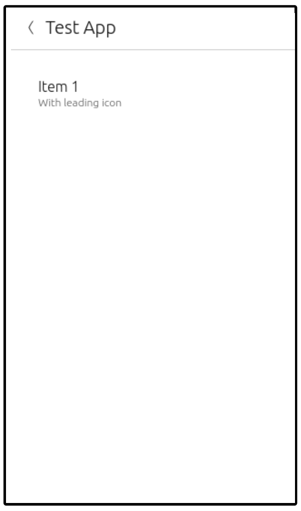
- One leading slot (start side)
- Up to two trailing slots (end side)
- Adjusts layout based on LTR / RTL locale

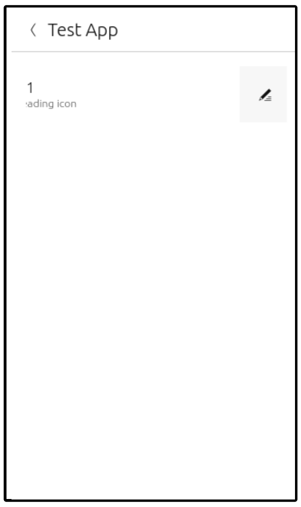
ViewItems Attached Properties - Defines behaviors such as expansion and dragging that affect swipe actions

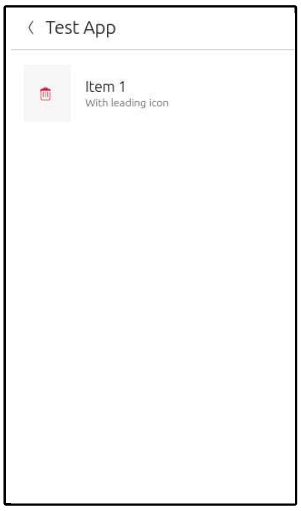
- expansionFlags - Attached property that controls the expandable behavior of a ListItem, defining how and when the item can expand or collapse within the list.
- dragMode - Attached property that specifies the drag behavior of a ListItem, determining how the item responds to drag gestures (e.g., for reordering or swipe interactions).
- collapseOnOutsideClick - Attached property that, when enabled, causes an expanded or swiped ListItem to collapse automatically if the user clicks or taps outside of it.

1.2.4 Pictures

The following three pictures, show some results of the discussed code.







1.2.5 Video

Use the link in this qr code to watch the video, you will see how the Leading and Trailing Actions work. When swiping a list item to the left or right, icons appear allowing the user to perform specific actions on that item.

1.3 Exercise

Now, it’s your turn to practice.

Implement leading and trailing actions on a list item in your project, then test it on the emulator or device to ensure the icons appear correctly when swiping.

1.4 References

https://ubports.gitlab.io/docs/api-docs/lomiriuserinterfacetoolkit/qml-lomiri-components-actionbar.html

Swipe up Gestures 2

2.1 Introduction

Swipe gestures are a core part of the Ubuntu Touch user experience, allowing users to navigate apps smoothly without relying on buttons. Among them, the Swipe Up gesture is commonly used to trigger actions such as opening menus, switching views, revealing hidden content, or moving between pages within an app.
In this topic, we will learn how to implement Swipe Up gestures in an Ubuntu Touch application, understand where they are best used, and explore how to handle user interactions effectively to provide a clean and intuitive mobile experience.

2.2 Swipe Up Gestures

2.2.1 SwipeArea

SwipeArea is a QML component designed to detect single-finger axis-aligned drag gestures (swipes). It is primarily used to implement gestures such as:

1. Swipe up to reveal a panel or drawer
2. Swipe down to hide a control
3. Swipe left/right for navigation

`SwipeArea` detects gestures within its own area, so placement and size are important.

Terminology

To understand Swipe Up Gestures, it's important to get familiar with a few basic terms:

1. `Swipe` - A finger movement across the screen in a specific direction.
2. `Drag` - The movement of a touch point across the screen, often recognized as a gesture.
3. `Gesture Recognition` - The process of detecting the type, direction, and length of a swipe.
4. `Edge Gesture` - A swipe starting from the screen edge, often reserved by the system (e.g., bottom-edge swipe to open app drawer).
5. `Drag Distance` - The total movement (in pixels or units) of a swipe from start to current position.
6. `Dragging Property` - A boolean indicating whether the swipe is currently happening.

Key Properties of SwipeArea

1. `Direction` - Specifies the allowed swipe direction. Options: `Upwards`, `Downwards`, `Leftwards`, `Rightwards`, `Vertical`, `Horizontal`.
2. `Distance` - Reports the distance of the swipe from start to current touch point.
3. `Dragging` - Becomes true when a swipe gesture is in progress.
4. `Grab Gesture` - If true, the SwipeArea takes full control of the gesture, preventing other components from reacting to it. Defaults to true.
5. `Immediate Recognition` - If true, the gesture is recognized as soon as touch lands. Defaults to false.
6. `Pressed` - true while the swipe area is being pressed.
7. `Touch Position` - Position of the touch relative to the SwipeArea.

Signals

`SwipeArea` emits signals to allow developers to react to swipe gestures:

1. *`onDraggingChanged`* - Triggered whenever dragging starts or stops. Useful to initiate actions when the swipe begins or ends.
2. *`onDistanceChanged`* - Triggered whenever distance changes. Useful for updating animations or UI feedback in real-time.

Explanation

1. *`Direction Control`*: direction ensures only swipes in the chosen direction are detected. For Swipe Up, use `SwipeArea.Upwards`.
2. *`Distance Tracking`*: distance provides how far the finger moved - this is useful to trigger actions only when the swipe passes a certain threshold.
3. *`Gesture Ownership`*: `grabGesture` ensures your `SwipeArea` receives the swipe exclusively. If set to false, other components might also respond.
4. *`Visual Feedback`*: Combine dragging and distance with QML animations to create interactive UI elements, e.g., sliding panels or “pull-up” menus.
5. *`Edge Placement`*: Often, the `SwipeArea` is placed at the bottom of the screen (`anchors.bottom`) with a small height, so it detects upward swipes starting from the screen edge.

Code Example:

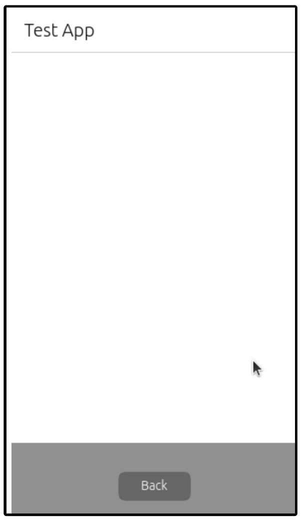
```
import QtQuick 2.4
import Lomiri.Components 1.3

MainView {
    width: units.gu(40)
    height: units.gu(70)

    Page {
        header: PageHeader {
            title: "Swipe Up Gesture Example"
        }

        Rectangle {
            anchors.left: parent.left
            anchors.right: parent.right
            anchors.bottom: parent.bottom
            height: units.gu(10)
            color: Qt.rgba(0.13, 0.13, 0.13, 0.5)

            MouseArea {
                id: swipeArea
                anchors.fill: parent
                property real startY: 0

                onPressed: {
                    startY = mouse.y
                }
                onReleased: {
                    if (startY - mouse.y > 40) {
                        console.log("Swipe Up Detected!")
                        panel.visible = true
                    }
                }
            }
        }

        Rectangle {
            id: panel
            width: parent.width
            height: units.gu(20)
            color: "#444"
            anchors.bottom: parent.bottom
            visible: false

            Label {
                anchors.centerIn: parent
                text: "Panel Opened with Swipe Up!"
                color: "white"
            }
        }
    }
}
```

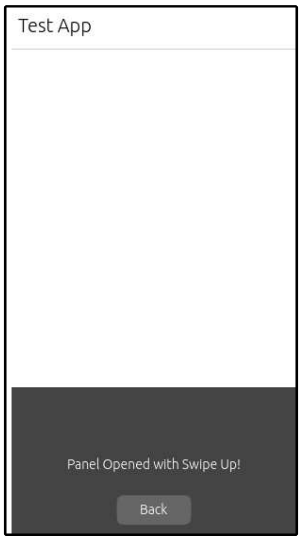
Explanation of Example:

1. `SwipeArea` at the bottom detects upward swipes.
2. `distance` tracks the finger movement.
3. If swipe passes `threshold`, panel appears.
4. `dragging` tracks the ongoing swipe.

2.3 Pictures

The following two pictures, show some results of the discussed code.





2.4 Video

The qr code links to this video, where you will see how the Swipe Up gesture works, where swiping from the bottom of the screen smoothly reveals a panel containing text.

2.5 Exercise

Time to practice.

Implement the Swipe Up gesture in your app to reveal the text panel, then test it on the emulator or device to ensure the bottom drawer opens smoothly.

2.6 References

SwipeArea Documentation[1]

[1] https://ubports.gitlab.io/docs/api-docs/lomiriuserinterfacetoolkit/qml-lomiri-components-swipearea.html

Multi Pane View for Convergence 3

3.1 Introduction

In Lomiri (Ubuntu Touch), applications can adapt their layout to different screen sizes, ensuring a consistent user experience across phones, tablets, and desktops. This concept is known as *Convergence*, and one of its key UI patterns is the *Multi-Pane View*.

With Multi-Pane View, an app can display a single screen on smaller devices, while automatically switching to a two-pane or multi-section layout on larger screens.

In this lesson, you will learn how to implement adaptive layouts using components such as `AdaptivePageLayout` and `PageColumnsLayout`, enabling your interface to transition smoothly from a single-pane view to a multi-pane view based on the available screen space.

3.2 Multi Pane View for Convergence

3.2.1 Introduction

In this lesson, you will learn how to create adaptive multi-pane layouts in QML that automatically adjust based on screen size. This convergence design principle allows the same application to work seamlessly across phones, tablets, and desktops by adapting the layout to the available screen space.

Multi-Pane View for Convergence is implemented using Adaptive Layouts such as:

- `AdaptivePageLayout`
- `PageStack + Page`
- `Column / Row` with responsive anchors

You will learn how to use `AdaptivePageLayout` with `PageColumnsLayout` to create master-detail interfaces that switch between single-pane (phone) and multi-pane (tablet/desktop) layouts automatically. This enables developers to build one application that provides an optimal user experience across all device types.

3.2.2 Terminology

1. *Convergence:* A design principle where the same app works across phones, tablets, and desktops with adaptive layouts.
2. *Single Pane:* Layout showing one page at a time (phone-style).
3. *Dual/Multi Pane:* Layout showing multiple sections (e.g., sidebar + content) simultaneously (tablet/desktop).

4. *Master-Detail:* Common multi-pane pattern: a list (master) on the left and details (content) on the right.
5. *AdaptivePageLayout:* Lomiri component that automatically switches between single/multi-pane layouts based on screen size.

3.2.3 Key Component: AdaptivePageLayout

AdaptivePageLayout is central to convergence. It manages multiple panes and adapts automatically.

3.2.4 Properties

Table 3.1: Properties

Property	*Type*	*Description*
primaryPage	Page	The main page (always visible in single-pane mode).
layouts	array	Array of PageColumnsLayout objects that define when and how multi-pane layout is activated.
columns	int	Number of columns currently displayed (1 for single-pane , 2+ for multi-pane).

3.2.5 PageColumnsLayout Properties

Table 3.2: PageColumnsLayout Properties

Property	*Type*	*Description*
when	Bool	Condition that triggers this layout (e.g. width > units.gui(80)).
PageColumn	object	Defines a column in the multi-pane layout with width constraints.

3.2.6 PageColumn Properties

Table 3.3: PageColumn Properties

Property	*Type*	*Description*
minimumWidth	real	Minimum width for the column (e.g, units.gui(30)).
maximumWidth	real	Maximum width for the column (e.g, units.gui(50)).
preferredWidth	real	Preferred width for the column (e.g, units.gui(40)).
fillWidth	bool	if true, the column fills remaining available width.

3.2.7 Methods

Method	*Description*
adPageToNextColumn(SourcePage, pageComponent properties)	Adds a new page to the next column with optional properties.
removePages(page)	Removes the specified page and returns to the previous column.

3.2.8 Recommended Sizes

Property	*Type*	*Single-Pane*	*Multi-Pane*
primaryPage	Page	Full Width (40-60 gu)	Sidebar (25-40 gu)
PageColumn (first)	PageColumn	Hidden	minimunWidth: 30 gu, maximumWidth: 50 gu, preferredWidth: 40 gu.
PageColumn (second)	PageColumn	Hidden	Content area with fillWidth: true (40-80+ fu)
when condition	bool	N/A below threshold	width > 80 gu triggerpoint
columns	int	1	2+

Table 3.4: Recommended Sizes

3.2.9 Convergence in Practice

- *Phone* –> Users navigate with stacked pages.
- *Tablet/Desktop* –> Same app shows multiple panes at once.
- *Developers don't need two apps* –> the layout adapts automatically.

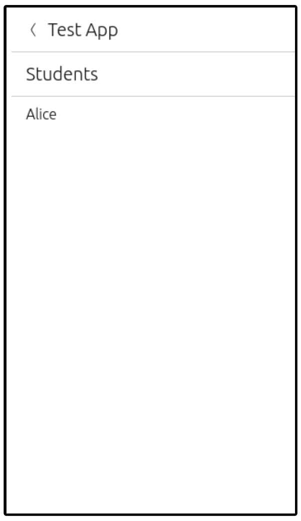
3.2.10 Use Cases

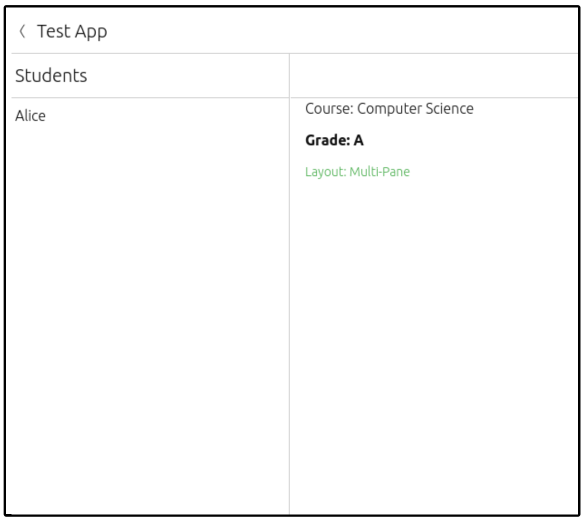
- *Email client* –> Inbox (left) + Email content (right)
- *Project manager* –> Project list (left) + Task board (right)
- *Settings app* –> Categories (left) + Detail settings (right)

3.3 Code Example

```
import QtQuick 2.4
import Lomiri.Components 1.3

Item {
    anchors.fill: parent

    AdaptivePageLayout {
        id: pageLayout
        anchors.fill: parent

        // When screen width > 80 gu, show 2 panes side by side
        layouts: [
            PageColumnsLayout {
                when: width > units.gu(80)
                PageColumn {
                    minimumWidth: units.gu(30)
                    maximumWidth: units.gu(50)
                    preferredWidth: units.gu(40)
                }
                PageColumn {
                    fillWidth: true
                }
            }
        ]

        // LEFT PANE: List with one student
        primaryPage: Page {
            id: listPage
            header: PageHeader {
                title: "Students"
            }

            ListView {
                width: parent.width
                height: parent.height - listPage.header.height
                anchors.bottom: parent.bottom
                clip: true
                spacing: 0
                model: ListModel {
                    ListElement { name: "Alice"; course: "Computer
    Science"; grade: "A" }
                }

                delegate: ListItem {
                    width: parent.width
                    height: units.gu(5)
                    onClicked: {
                        pageLayout.addPageToNextColumn(listPage,
    detailPageComponent, {
                            studentName: name,
                            studentCourse: course,
                            studentGrade: grade
                        })
                    }

                    Label {
                        anchors.left: parent.left
                        anchors.leftMargin: units.gu(2)
                        anchors.verticalCenter: parent.verticalCenter
                        text: name
                        font.pixelSize: units.gu(2)
                    }
                }
            }
        }
    }

    // RIGHT PANE: Detail view
    Component {
        id: detailPageComponent
        Page {
            id: detailPage
            header: PageHeader {
                title: pageLayout.columns === 1 ? "Student Details" : "
     "
                leadingActionBar.actions: [
                    Action {
                        iconName: "back"
                        text: "Back"
                        visible: pageLayout.columns === 1
                        onTriggered: pageLayout.removePages(detailPage)
                    }
                ]
            }

            property string studentName: ""
            property string studentCourse: ""
            property string studentGrade: ""

            Column {
                anchors.top: parent.top
                anchors.left: parent.left
                anchors.right: parent.right
                anchors.margins: units.gu(2)
                anchors.topMargin: units.gu(2)
                spacing: units.gu(2)
                width: parent.width - units.gu(4)

                Label {
                    text: "Name: " + (studentName || "Select a student"
     )
                    font.pixelSize: units.gu(2)
                    width: parent.width
                }

                Label {
                    text: "Course: " + (studentCourse || "-")
                    font.pixelSize: units.gu(2)
                    width: parent.width
                }

                Label {
                    text: "Grade: " + (studentGrade || "-")
                    font.pixelSize: units.gu(2)
                    font.bold: true
                    width: parent.width
                }

                Label {
                    text: "Layout: " + (pageLayout.columns > 1 ? "Multi
     -Pane" : "Single-Pane")
                    font.pixelSize: units.gu(1.8)
                    color: pageLayout.columns > 1 ? "#4CAF50" : "#
     FF9800"
                    width: parent.width
                }
            }
        }
    }
}
```

3.4 Pictures





Course: Computer Science

Grade: A

Layout: Multi-Pane

3.5 Video

The following qr code links to the following video, you will see how the Responsive Convergence works, where the layout dynamically adapts from a multi-pane view to a single-pane view as the window is resized.

3.6 Exercise

Your turn to build.

Implement a multi-page navigation structure that supports convergence, then resize the app window or rotate the device to confirm the layout adapts correctly.

3.7 References

1. AdaptivePageLayout QML Type[I]
2. Timesheet App - Github[II]

[I] https://ubports.gitlab.io/docs/api-docs/lomiriuserinterfacetoolkit/qml-lomiri-components-adaptivepagelayout.html#: :text=AdaptivePageLayout%20QML%20Type

[II] https://github.com/CITOpenRep/timemanagement/blob/9c5c2efcf8219b89fea9d0d444915bfb8eedb41e/qml/TSApp.qmlŁ126C5-L361C1

Floating Action Button (FAB) 4

4.1 Introduction

A *Floating Action Button (FAB)* is a prominent UI element used in mobile applications to highlight the primary action on a screen. It is designed to be highly visible and easily accessible, enabling users to perform key actions quickly, especially in frequently used workflows.

A FAB is typically positioned at the bottom-right corner of the screen, where it can be easily reached with the thumb. While its appearance may vary depending on the application design, it is most commonly displayed as a circular button (or a rounded square) containing an icon.

The main purpose of a FAB is to provide a direct shortcut for actions such as creating, adding, or starting something, without requiring users to search through menus.

In this lesson, you will learn how to implement a Floating Action Button in an Ubuntu Touch application and connect it to functional actions such as opening a new page, triggering a dialog, or initiating a workflow within the app.

4.2 Floating Action Button

In Lomiri QML, FABs can be created using:

1. `ActionButton` from `Lomiri.Components`
2. A *custom circular*`Button` with Icon
3. Adding *FAB Menu* via a column/stack of smaller FABs revealed on tap

4.2.1 Terminology

1. FAB - Floating Action Button, circular primary action button
2. ActionButton - Lomiri component for buttons
3. IconName - Name of the symbolic icon used inside the button
4. FAB Menu - A group of FABs shown when the main FAB is expanded
5. Overlay Layer - FABs usually float above page content

4.2.2 Key Properties

1. `iconName` - Icon to display inside the button (e.g., ”add”)
2. `color` - Background color of the FAB
3. `anchors` - Usually anchored bottom & right
4. `onClicked` - Defines what happens when the button is pressed

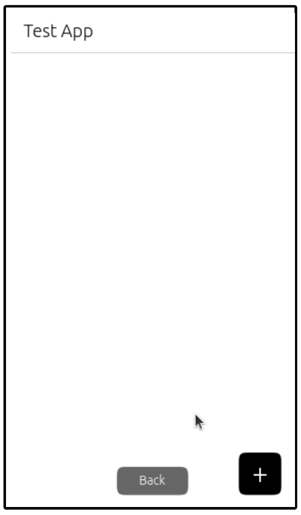
4.2.3 Explanation

1. mainFab –> The primary FAB (toggles menu open/close).
2. menuOpen property –> Controls whether the mini-FABs are shown.
3. Sub FABs (Timesheet, Task, Activity) –> Appear stacked above the main FAB when open.
4. Animation (Behavior on opacity) –> Smooth fade-in/out effect.

4.2.4 Example Code

```

import QtQuick 2.4
import Lomiri.Components 1.3

MainView {
    width: units.gu(40)
    height: units.gu(70)

    Page {
        id: page
        title: "FAB Example"

        property bool showMessage: false

        Label {
            id: messageLabel
            text: "Fab Clicked"
            anchors.centerIn: parent
            font.pixelSize: units.gu(3)
            visible: page.showMessage
        }

        Button {
            id: fab
            text: "+"
            anchors.right: parent.right
            anchors.bottom: parent.bottom
            anchors.margins: units.gu(2)
            width: units.gu(6)
            height: units.gu(6)
            font.pixelSize: units.gu(4)
            onClicked: {
                console.log("FAB Clicked - Create new item")
                page.showMessage = true
            }
        }
    }
}
```

4.2.5 Pictures



4.2.6 Video

If you use this qr code, it will point you to this video, you will see how the Floating Action Button (FAB) works, where tapping the rounded square plus button displays a 'FAB CLICKED' message on the screen.

4.3 Exercise

Give this a try.

Add a Floating Action Button to your screen's corner, then tap it on the emulator or device to verify the action triggers correctly.

4.4 References

Timesheet App - Github[1]

[1] https://github.com/CITOpenRep/timemanagement/blob/9c5c2efcf8219b89fea9d0d444915bfb8eedb41e/qml/Timesheet_Page.qml#L285C5-L312C1

Option Selector 5

5.1 Introduction

An Option Selector is a user interface component that allows users to choose one option from a list of multiple options. Although several choices may be available, the user can select only one at a time, ensuring a clear and unambiguous selection. This is commonly used in applications when a single confirmed choice is required, such as selecting a category, mode, account, or setting.

In this lesson, you will learn how to implement an Option Selector in an Ubuntu Touch application, configure the available options, and handle the selected value within your app logic. This will help you build user-friendly screens where users can make quick and reliable selections.

5.2 Option Selector

In Ubuntu Touch (Lomiri) apps, the `OptionSelector` component is used to let users select a single option from a list. It's similar to a drop-down menu but optimized for touch interaction and small screens. The selector can display a list of options, highlight the current selection, and trigger actions when the user changes their choice.

5.2.1 Terminology

1. *OptionSelector* — A Lomiri component that provides a touch-friendly interface for choosing one option from a list.
2. *Model* — The data source containing the selectable options.
3. *SelectedIndex* — The index of the currently selected item.
4. *SelectedValue* — The text or data of the currently selected item.
5. *Delegate* — Defines how each item is visually represented in the selector.

5.2.2 Key Properties

Table 5.1: Key Properties

Property	*Type*	*Description*
model	list / model	List of options for selection
currentIndex	int	Index of the currently selected option
currentText	string (read-only)	Text of the currently selected option.
delegate	Component	Defines how each item is rendered in the dropdown.
editable	bool	If true, user can type to filter or enter new values.
popup	Item	The dropdown popup displayed when selecting an option.

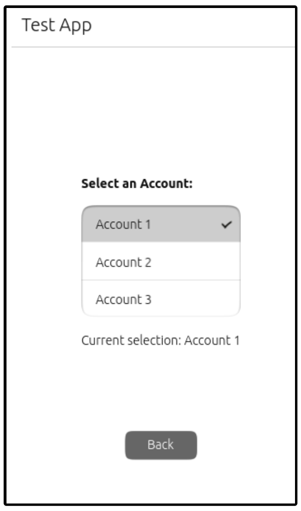
5.2.3 Signals

1. *onSelectedIndexChanged* — Triggered when the selected index changes.
2. *onSelectedValueChanged* — Triggered when the selected value changes.

5.2.4 Code Example

```
import QtQuick 2.7
import Lomiri.Components 1.3

// Option Selector / Combo Selector Demo
MainView {
    width: units.gu(40)
    height: units.gu(20)
    objectName: "mainView"

    property var options: [
        "Account 1",
        "Account 2",
        "Account 3"
    ]

    Column {
        anchors.centerIn: parent
        spacing: units.gu(2)

        Label {
            text: "Select an Account:"
            font.bold: true
        }

        OptionSelector {
            id: comboSelector
            model: options
            anchors.horizontalCenter: parent.horizontalCenter
            onSelectedIndexChanged: {
                // Handle selection change
                console.log("Selected index:", selectedIndex, "text:",
 options[selectedIndex])
            }
        }

        Label {
            text: "Current selection: " + options[comboSelector.
 selectedIndex]
        }
    }
}
```

5.3 Pictures



5.4 Video

The following qr code links to the following video, you will see how the Option Selector works, where choosing an account from the list updates the display with the selected account name and number.

5.5 Exercise

Put it into practice.

Implement an Option Selector in your settings menu, then toggle between different choices on the emulator or device to verify the selection updates correctly.

5.6 References

1. Item Selector Documentation[1]

[1] https://ubports.gitlab.io/docs/api-docs/lomiriuserinterfacetoolkit/qml-lomiri-components-listitems-itemselector.html