helloQt is the Qt development
branch of Bergx2 GmbH

Fürstenstr. 15
80333 München
Germany

info@helloqt.de
https://www.helloqt.de

# Mastering screen management in Qt 6: A comprehensive guide

23 June 2024, by Lucas Moreira de Oliveira

Choosing the right approach for managing screens in a Qt 6 application is crucial for ensuring a smooth and efficient user experience. Whether you are developing a simple app with a few screens or a complex application with dynamic interfaces, understanding the various methods available in Qt 6 will help you make informed decisions. In this article, we explore different techniques for managing screens, delving into their implementations, advantages, and disadvantages, and the scenarios where each method shines.

### 1. Dynamic screen creation with `component.createComponent` and `component.createObject`

One of the powerful features of Qt 6 is its ability to create components dynamically at runtime. This is particularly useful when the number or type of screens is not known at compile time.

#### 1.1 Implementation example

```qml
import QtQuick 2.15
import QtQuick.Controls 2.15

ApplicationWindow {
    visible: true
    width: 640
    height: 480

    Button {
        text: "Create Screen"
        onClicked: createScreen()
    }
    function createScreen() {
        var component = Qt.createComponent("Screen.qml");
        if (component.status === Component.Ready) {
            var dynamicScreen = component.createObject(applicationWindow);
            dynamicScreen.width = 640;
            dynamicScreen.height = 480;
        }
    }
    function destroyScreen(screen) {
        if (screen !== null) {
            screen.destroy();
```

```
        }
    }
}
```

In this example, a new screen is created dynamically when the button is clicked. The `Qt.createComponent` function loads the QML component and `createObject` instantiates it.

### 1.3 Advantages and disadvantages of dynamic screen creation


Dynamic creation offers flexibility and efficient memory usage, as components are only instantiated when needed. However, it can introduce a slight delay during creation and requires careful error handling to manage the component's loading state and potential failures.

This method is ideal for scenarios where screens are not needed simultaneously, such as in multi-step forms or feature-rich applications with many optional modules.

### 1.4 Further reading

- Dynamic QML Object Creation from JavaScript: https://doc.qt.io/qt-6/qtqml-javascript-dynamicobjectcreation.html

### 2. Screen management with `Loader`

The `Loader` element in Qt 6 provides a convenient way to manage the loading and unloading of QML components, enabling lazy loading and reducing the initial memory footprint of your application.

### 2.1 Implementation example

```qml
import QtQuick 2.15
import QtQuick.Controls 2.15

ApplicationWindow {
    visible: true
    width: 640
    height: 480

    Loader {
        id: screenLoader
        anchors.fill: parent
    }
```

```
    Button {
        text: "Load Screen"
        onClicked: screenLoader.source = "Screen.qml"
    }
}
```

Here, a `Loader` is used to load a screen when the button is clicked. The `Loader` can also unload the component by setting its `source` property to an empty string.

### 2.2 Advantages and disadvantages of the loader in Qt 6

Using `Loader` simplifies memory management and improves performance by only loading components when needed. However, frequent loading and unloading can lead to performance bottlenecks, especially if the components are complex.

`Loader` is best suited for applications where screens are used intermittently and do not need to be kept in memory once they are no longer visible.

### 2.3 Further reading

- Qt documentation on Loader: https://doc.qt.io/qt-6/qml-qtquick-loader.html

- Lazy loading in Qt Quick: https://doc.qt.io/qt-6/qtquick-performance.html#lazy-loading

## 3. Visibility based screen switching

Another straightforward approach is to create all screens at once and manage their visibility using the `visible` property.

### 3.1 Implementation example

```
```qml
import QtQuick 2.15
import QtQuick.Controls 2.15

ApplicationWindow {
    visible: true
    width: 640
    height: 480

    Rectangle {
        id: screen1
        widt


h: parent.width
        height: parent.height
        color: "red"
        visible: true
```
```

```
    }

    Rectangle {
        id: screen2
        width: parent.width
        height: parent.height
        color: "blue"
        visible: false
    }

    Button {
        text: "Switch Screen"
        onClicked: {
            screen1.visible = !screen1.visible;
            screen2.visible = !screen2.visible;
        }
    }
}
```

In this example, two screens are created and their visibility is toggled with a button click.

### 3.2 Advantages and disadvantages oft he visible property

Managing screens with visibility is simple and quick to implement. However, it consumes more memory since all screens are kept in memory at all times, which can be a drawback for applications with many screens or resource-intensive components.

This approach is suitable for small to medium-sized applications where the overhead of keeping all screens in memory is negligible.

### 3.3 Further reading

- Qt documentation on Property Binding: https://doc.qt.io/qt-6/qtqml-syntax-propertybinding.html

## 4. Stack-based navigation with `StackView`

`StackView` provides a stack-based navigation model, where screens are pushed and popped from the stack, mimicking a typical navigation pattern found in mobile applications.

### 4.1 Implementation example

```
```qml
import QtQuick 2.15
import QtQuick.Controls 2.15

ApplicationWindow {
    visible: true
```
```

```
    width: 640
    height: 480

    StackView {
        id: stackView
        anchors.fill: parent
        initialItem: Screen1 {}
    }

    Button {
        text: "Push Screen"
        onClicked: stackView.push(Qt.resolvedUrl("Screen2.qml"))
    }
}
```

Here, a `StackView` manages the navigation between screens. The `initialItem` property sets the first screen, and additional screens are pushed onto the stack with the `push` method.

### 4.2 Advantages and disadvantages of StackView

`StackView` provides an intuitive way to manage navigation and maintains a clear history of screens, which is useful for back navigation. However, managing complex navigation stacks can become cumbersome and might impact performance if not handled properly.

This method is recommended for applications with hierarchical navigation structures, such as settings menus or multi-step processes, where back navigation is required.

### 4.3 Further reading

- Qt documentation on StackView: https://doc.qt.io/qt-6/qml-qtquick-controls-stackview.html

- Implementing navigation with StackView: https://doc.qt.io/qt-6/qtquickcontrols2-navigation.html

## 5. Swipe-based navigation with `SwipeView`

`SwipeView` provides a touch-friendly navigation model, ideal for mobile applications or touch interfaces where users can swipe between different screens.

### 5.1 Implementation example

```qml
import QtQuick 2.15
import QtQuick.Controls 2.15

ApplicationWindow {
    visible: true
    width: 640
```

```
    height: 480

    SwipeView {
        anchors.fill: parent
        Rectangle { color: "red"; width: parent.width; height: parent.height }
        Rectangle { color: "blue"; width: parent.width; height: parent.height
}
        Rectangle { color: "green"; width: parent.width; height: parent.height
}
    }
}
```

In this example, `SwipeView` is used to allow users to swipe between three different screens, represented as `Rectangle` components with different colours.

### 5.2 Advantages and disadvantages of SwipeView

`SwipeView` is highly intuitive and easy to use, especially for mobile applications. However, it is less suited for desktop applications where swiping is not a common interaction method. `SwipeView` is ideal for mobile applications, image galleries, or any interface where quick and natural navigation between screens is desired.

### 5.3 Further reading

- Qt Documentation on SwipeView: https://doc.qt.io/qt-6/qml-qtquick-controls-swipeview.html

## 6. Tab-based navigation with `TabBar`

`TabBar` provides a tabbed navigation model, where users can switch between different tabs, each representing a different screen or section of the application.

### 6.1 Implementation example

```
import QtQuick 2.15
import QtQuick.Controls 2.15

ApplicationWindow {
    visible: true
    width: 640
    height: 480

    TabBar {
        id: tabBar
        width: parent.width
        TabButton {
            text: "Red"
            onClicked: stackView.replace(Qt.resolvedUrl("RedScreen.qml"))
        }
        TabButton {
```

```
            text: "Blue"
            onClicked: stackView.replace(Qt.resolvedUrl("BlueScreen.qml"))
        }
        TabButton {
            text: "Green"
            onClicked: stackView.replace(Qt.resolvedUrl("GreenScreen.qml"))
        }
    }

    StackView {
        id: stackView
        anchors.top: tabBar.bottom
        anchors.left: parent.left
        anchors.right: parent.right
        anchors.bottom: parent.bottom
        initialItem: RedScreen {}
    }
}
```

In this example, `TabBar` is used in conjunction with `StackView` to create a tabbed interface. Each tab button replaces the current screen in the `StackView` with a new one.

### 6.2 Advantages and disadvantages of TabBar

`TabBar` offers a familiar and easy-to-use navigation model, making it ideal for desktop applications or any application with clearly defined sections. However, it may not be as suitable for applications with a large number of screens or for those requiring a more dynamic navigation flow. `TabBar` is perfect for settings pages, dashboards, or applications with a few primary sections.

### 6.3 Further reading

- Qt Documentation on TabBar: https://doc.qt.io/qt-6/qml-qtquick-controls-tabbar.html

- Qt Documentation on TabButton: https://doc.qt.io/qt-6/qml-qtquick-controls-tabbutton.html

## 7. Conclusion

Choosing the right screen management approach in Qt 6 depends on the specific needs of your application. Dynamic creation with `component.createComponent` and `component.createObject` offers flexibility and efficiency for dynamic interfaces. `Loader` provides a simple way to manage memory usage by loading screens on demand. Visibility-based switching is quick to implement but can consume more memory. `StackView` is ideal for applications with hierarchical navigation. `SwipeView` and `TabBar` offer intuitive navigation models for mobile and desktop applications, respectively.

Understanding these methods and their trade-offs will help you build more efficient and responsive Qt 6 applications, tailored to the unique requirements of your projects.