# Fyne Framework - Complete Developer Guide

**Author:** GitHub Copilot
**Date:** December 1, 2025
**Project:** SkillDar Client Application

---

## Table of Contents



---

## Introduction

Fyne is a cross-platform GUI toolkit for Go that makes it easy to build beautiful, native applications for desktop and mobile platforms.

### Key Principles

1. **Widgets = Interactive**: Buttons, entries, checkboxes
2. **Canvas = Visual**: Rectangles, circles, images
3. **Containers = Layout**: Automatically arrange children
4. **Theme = Appearance**: Colors, fonts, sizes
5. **Refresh = Update**: Changes need refresh to display

### The Mental Model (LEGO Blocks)

```
Level 1: Pre-made blocks (Widgets)
  ↓ Use these 90% of the time
  widget.Button, widget.Label, widget.Entry
```

```
Level 2: Visual blocks (Canvas)
  ↓ For custom visuals
  canvas.Rectangle, canvas.Circle, canvas.Text

Level 3: Organizers (Containers)
  ↓ Arrange your blocks
  container.NewVBox, container.NewHBox, container.NewStack

Level 4: Custom blocks (When needed)
  ↓ Only when Level 1-3 aren't enough
  Make your own widget (like ThemedNavBar)
```

---

## Core Architecture

### Application Lifecycle

The app is the root of everything - it manages windows, settings, and the event loop.

```go
// Create the application instance
app := app.New()

// The app manages:
// - Theme settings
// - All windows
// - Application preferences/storage
// - Lifecycle events (quit, resume, etc.)

// Access app from anywhere
fyne.CurrentApp()

// Set application-wide theme
app.Settings().SetTheme(myCustomTheme)

// Get/set preferences
app.Preferences().SetString("username", "john")
username := app.Preferences().String("username")
```

### Windows

Windows are top-level containers that hold your UI.

```
// Create a window
w := app.NewWindow("My App")

// Window properties
w.Resize(fyne.NewSize(800, 600))    // Set size
w.SetMaster()                        // Main window (app closes when
w.CenterOnScreen()                   // Position at screen center
w.SetFixedSize(true)                 // Prevent resizing
w.SetFullScreen(true)                // Full screen mode

// Content is what's displayed
w.SetContent(myContainer)

// Show and run (blocks until window closes)
w.ShowAndRun()  // Use for main window

// Or just show (non-blocking)
w.Show()        // Use for secondary windows
```

**Critical**: The window holds ONE CanvasObject. Change content by calling `SetContent()` again.

---

## Widgets

Widgets are smart objects that:

- Handle user interaction (clicks, typing, dragging)
- Manage their own state
- Automatically update appearance
- Respond to theme changes

### Common Widgets

#### Label - Display Text

```
label := widget.NewLabel("Hello World")
label.TextStyle = fyne.TextStyle{Bold: true, Italic: true}
label.Alignment = fyne.TextAlignCenter
label.Wrapping = fyne.TextWrapWord  // Wrap long text
label.Text = "Updated"  // Change text
label.Refresh()         // Must call to see changes!
```

### Button - Clickable Action

```
btn := widget.NewButton("Click Me", func() {
    fmt.Println("Button clicked!")
})
btn.Importance = widget.HighImportance  // Primary action (blue)
btn.Importance = widget.LowImportance   // Secondary action (subtle)
btn.Disable()   // Make unclickable
btn.Enable()    // Make clickable again
```

### Entry - Text Input

```
entry := widget.NewEntry()
entry.SetPlaceHolder("Type here...")
entry.OnChanged = func(text string) {
    fmt.Println("Text changed:", text)
}
entry.Validator = func(s string) error {
    if len(s) < 5 {
        return errors.New("too short")
    }
    return nil
}
password := widget.NewPasswordEntry()  // Masked input
```

### Checkbox

```
check := widget.NewCheck("Enable feature", func(checked bool) {
    fmt.Println("Checked:", checked)
})
check.SetChecked(true)
```

### Select - Dropdown Menu

```
sel := widget.NewSelect([]string{"Option 1", "Option 2"}, func(value
    fmt.Println("Selected:", value)
})
sel.SetSelected("Option 1")
```

### Form - Group Inputs with Labels

```
form := widget.NewForm(
    widget.NewFormItem("Name", nameEntry),
    widget.NewFormItem("Email", emailEntry),
)
form.OnSubmit = func() {
    fmt.Println("Form submitted")
}
```

### Widget Importance Levels

```
widget.HighImportance    // Primary action - stands out (blue button
widget.MediumImportance  // Default - normal appearance
widget.LowImportance     // Subtle - blends in (text-like button)
widget.DangerImportance  // Destructive action (red)
widget.WarningImportance // Caution (orange/yellow)
widget.SuccessImportance // Positive action (green)
```

---

## Canvas Objects

Canvas objects are low-level drawing primitives with NO built-in interaction.

### Text - Styled Text Rendering

```
text := canvas.NewText("Hello", color.White)
text.Alignment = fyne.TextAlignCenter
text.TextSize = 24
text.TextStyle = fyne.TextStyle{Bold: true}
```

### Rectangle - Filled Rectangle

```
rect := canvas.NewRectangle(color.RGBA{R: 100, G: 150, B: 200, A: 25
rect.SetMinSize(fyne.NewSize(200, 100))
rect.StrokeColor = color.Black  // Border color
rect.StrokeWidth = 2            // Border width
```

### Circle - Filled Circle

```
circle := canvas.NewCircle(color.Red)
circle.StrokeColor = color.Black
```

```
circle.StrokeWidth = 3
circle.Resize(fyne.NewSize(50, 50))
```

### Line - Straight Line

```
line := canvas.NewLine(color.Black)
line.StrokeWidth = 2
line.Position1 = fyne.NewPos(0, 0)
line.Position2 = fyne.NewPos(100, 100)
```

### Image - Display Image

```
// From file
img := canvas.NewImageFromFile("photo.jpg")
img.FillMode = canvas.ImageFillContain  // Maintain aspect ratio
img.FillMode = canvas.ImageFillStretch  // Stretch to fit
img.FillMode = canvas.ImageFillOriginal // Original size

// From resource (bundled)
img := canvas.NewImageFromResource(myResource)
```

### Key Difference

- **Canvas objects** = "dumb" visual elements
- **Widgets** = "smart" interactive components

To make canvas objects interactive, wrap them in a custom widget or container.

---

## Containers

Containers arrange multiple objects automatically. They're the secret to responsive UIs.

### VBox - Vertical Stack

```
vbox := container.NewVBox(
    widget.NewLabel("First"),
    widget.NewLabel("Second"),
    widget.NewLabel("Third"),
)
// Items stack vertically, each takes minimum height needed
```

HBox - Horizontal Stack

```
hbox := container.NewHBox(
    widget.NewButton("Left", nil),
    widget.NewButton("Center", nil),
    widget.NewButton("Right", nil),
)
// Items arrange horizontally, each takes minimum width needed
```

Spacer - Push Items Apart

```
container.NewHBox(
    widget.NewLabel("Left"),
    layout.NewSpacer(),  // This pushes items to edges
    widget.NewLabel("Right"),
)
```

Border - Place Items on Edges

```
border := container.NewBorder(
    topItem,     // Top edge (full width)
    bottomItem,  // Bottom edge (full width)
    leftItem,    // Left edge (remaining height)
    rightItem,   // Right edge (remaining height)
    centerItem,  // Center (fills remaining space)
)
// Pass nil for edges you don't need
```

Stack - Layer Items

```
stack := container.NewStack(
    background,  // Bottom layer
    content,     // Middle layer
    overlay,     // Top layer
)
// All items same size, layered like Photoshop layers
```

Center - Center Single Item

```
center := container.NewCenter(myWidget)
```

Padded - Add Padding

```
padded := container.NewPadded(myWidget)
// Adds equal padding on all sides
```

Grid - Fixed Columns

```
grid := container.NewGridWithColumns(3,
    item1, item2, item3,
    item4, item5, item6,
)
// 3 columns, rows auto-added as needed
```

GridWrap - Fixed Cell Size

```
gridWrap := container.NewGridWrap(
    fyne.NewSize(100, 80),  // Each cell is 100x80
    item1, item2, item3, item4, item5,
)
// Creates as many columns as fit, wraps overflow to new rows
```

Scroll - Scrollable Container

```
scroll := container.NewScroll(tallContent)
scroll.SetMinSize(fyne.NewSize(300, 200))
// Content bigger than min size becomes scrollable
```

Split - Divider

```
// Vertical divider (user can drag)
vsplit := container.NewVSplit(leftPanel, rightPanel)
vsplit.SetOffset(0.3)  // Left panel takes 30% width

// Horizontal divider
hsplit := container.NewHSplit(topPanel, bottomPanel)
```

### Container Behavior

```
// Containers automatically:
// 1. Calculate sizes based on content
// 2. Position children
// 3. Handle resizing
// 4. Refresh when content changes

// Update container contents
myContainer.Objects = []fyne.CanvasObject{newItem1, newItem2}
myContainer.Refresh()

// Add/remove items (VBox/HBox only)
myContainer.Add(newWidget)
myContainer.Remove(oldWidget)
```

---

## Layout System

Understanding the layout system is crucial for building UIs correctly.

### Size Constraints

```
// MINIMUM SIZE - Smallest size object wants to be
widget.SetMinSize(fyne.NewSize(200, 50))

// Objects declare their MinSize():
// - Labels: based on text length
// - Buttons: based on text + padding
// - Containers: based on children

// SIZE vs MinSize
obj.Size()     // Current size (set by layout)
obj.MinSize()  // Minimum preferred size

// RESIZE - Try to change size
obj.Resize(fyne.NewSize(300, 100))
// ⚠️ Only works if object is NOT in a container
// Containers ignore manual Resize() and calculate size automaticall
```

### Layout Flow

```
1. Container calculates MinSize from children
   ↓
2. Container receives size from parent
   ↓
3. Container distributes space to children
   ↓
4. Children render at assigned size
```

Example

```
// This WON'T work (button in container ignores Resize):
btn := widget.NewButton("Click", nil)
btn.Resize(fyne.NewSize(500, 100))  // ❌ Ignored!
container.NewVBox(btn)

// This WILL work (button determines its own size):
btn := widget.NewButton("Very Long Button Text Here")
// Button will be wide enough to fit text
```

---

## Theming System

Themes control ALL visual aspects: colors, fonts, sizes, icons.

### Theme Interface

```
type MyTheme struct {
    variant fyne.ThemeVariant  // Light or Dark
}

// COLOR - Return color for named element
func (t MyTheme) Color(name fyne.ThemeColorName, variant fyne.ThemeV
    switch name {
    case theme.ColorNamePrimary:
        return color.RGBA{0x28, 0x7D, 0xF7, 0xFF}  // Brand color
    case theme.ColorNameBackground:
        if variant == theme.VariantDark {
            return color.RGBA{0x1A, 0x1A, 0x1A, 0xFF}  // Dark bg
        }
        return color.RGBA{0xFF, 0xFF, 0xFF, 0xFF}  // Light bg
    case theme.ColorNameForeground:
        // Text color
```

```
    case theme.ColorNameButton:
        // Button background
    case theme.ColorNameDisabled:
        // Disabled element color
    }
    // Fall back to default for unhandled colors
    return theme.DefaultTheme().Color(name, variant)
}

// SIZE - Return size for named element
func (t MyTheme) Size(name fyne.ThemeSizeName) float32 {
    switch name {
    case theme.SizeNameText:
        return 14  // Base text size
    case theme.SizeNameHeadingText:
        return 24  // Heading text size
    case theme.SizeNamePadding:
        return 4   // Padding between elements
    case theme.SizeNameInnerPadding:
        return 8   // Padding inside widgets
    }
    return theme.DefaultTheme().Size(name)
}

// FONT - Return font for text style
func (t MyTheme) Font(style fyne.TextStyle) fyne.Resource {
    if style.Bold {
        return myBoldFont
    }
    return myRegularFont
}

// ICON - Return icon resource
func (t MyTheme) Icon(name fyne.ThemeIconName) fyne.Resource {
    return theme.DefaultTheme().Icon(name)
}
```

Using Theme Colors

```
// ✅ CORRECT - Use theme colors
bg := canvas.NewRectangle(theme.BackgroundColor())

// ❌ WRONG - Hardcoded colors
```

```
bg := canvas.NewRectangle(color.White)  // Won't change with theme!

// Access current theme
currentTheme := fyne.CurrentApp().Settings().Theme()
primaryColor := currentTheme.Color(theme.ColorNamePrimary, theme.Var
```

### Theme Variants

```
theme.VariantLight  // Light mode
theme.VariantDark   // Dark mode

// Get current variant
variant := fyne.CurrentApp().Settings().ThemeVariant()

// Your theme can adapt to variant
func (t MyTheme) Color(name fyne.ThemeColorName, variant fyne.ThemeV
    if variant == theme.VariantDark {
        // Return dark colors
    } else {
        // Return light colors
    }
}
```

---

## Event Handling

### Widget Callbacks

```
// BUTTON - Single callback
btn := widget.NewButton("Click", func() {
    // Handle click
})

// ENTRY - Multiple events
entry := widget.NewEntry()
entry.OnChanged = func(text string) {
    // Fires on every keystroke
}
entry.OnSubmitted = func(text string) {
    // Fires when user presses Enter
}

// CHECK - State change
```

```
check := widget.NewCheck("Option", func(checked bool) {
    // Fires when check state changes
})
```

### Custom Interactive Widgets

```
// Implement Tappable interface
type ClickableCard struct {
    widget.BaseWidget
    content fyne.CanvasObject
    onTap   func()
}

func (c *ClickableCard) Tapped(*fyne.PointEvent) {
    if c.onTap != nil {
        c.onTap()
    }
}

func (c *ClickableCard) TappedSecondary(*fyne.PointEvent) {
    // Right click / long press
}

// Implement Hoverable for desktop
func (c *ClickableCard) MouseIn(*desktop.MouseEvent) {
    // Mouse entered
}

func (c *ClickableCard) MouseOut() {
    // Mouse left
}

func (c *ClickableCard) MouseMoved(*desktop.MouseEvent) {
    // Mouse moved over widget
}

// Change cursor
func (c *ClickableCard) Cursor() desktop.Cursor {
    return desktop.PointerCursor  // Hand cursor
}
```

---

## Data Binding

Data binding automatically syncs UI with data changes.

### String Binding

```
str := binding.NewString()
str.Set("Hello")

// Create widget bound to data
label := widget.NewLabelWithData(str)

// Update data - label updates automatically!
str.Set("World")  // Label now shows "World", no Refresh() needed!

// Get current value
value, _ := str.Get()

// Listen for changes
str.AddListener(binding.NewDataListener(func() {
    val, _ := str.Get()
    fmt.Println("String changed to:", val)
}))
```

### Other Binding Types

```
intData := binding.NewInt()
floatData := binding.NewFloat()
boolData := binding.NewBool()
```

### List Binding

```
list := binding.NewStringList()
list.Append("Item 1")
list.Append("Item 2")

listWidget := widget.NewListWithData(
    list,
    func() fyne.CanvasObject {
        return widget.NewLabel("")  // Template
    },
    func(item binding.DataItem, obj fyne.CanvasObject) {
        label := obj.(*widget.Label)
        str := item.(binding.String)
```

```
        val, _ := str.Get()
        label.SetText(val)
    },
)
```

### Struct Binding

```
type Person struct {
    Name string
    Age  int
}
data := binding.NewStruct(&Person{})
```

---

## Custom Widgets

Create reusable custom components when built-in widgets aren't enough.

### Basic Structure

```
// STEP 1: Define struct
type CustomCard struct {
    widget.BaseWidget  // ⚠️ MUST embed BaseWidget

    // Your data
    title string
    icon  fyne.Resource
}

// STEP 2: Constructor
func NewCustomCard(title string, icon fyne.Resource) *CustomCard {
    card := &CustomCard{
        title: title,
        icon:  icon,
    }
    card.ExtendBaseWidget(card)  // ⚠️ CRITICAL - Initialize base
    return card
}

// STEP 3: CreateRenderer - Define appearance
func (c *CustomCard) CreateRenderer() fyne.WidgetRenderer {
    // Build your widget's visual structure
    icon := canvas.NewImageFromResource(c.icon)
```

```
    title := widget.NewLabel(c.title)

    content := container.NewBorder(
        nil, nil,
        icon,
        nil,
        title,
    )

    return widget.NewSimpleRenderer(content)
}

// OPTIONAL: Add methods
func (c *CustomCard) SetTitle(title string) {
    c.title = title
    c.Refresh()  // Update appearance
}

// OPTIONAL: Implement interfaces for interaction
func (c *CustomCard) Tapped(*fyne.PointEvent) {
    fmt.Println("Card tapped!")
}
```

### Understanding ExtendBaseWidget

```
nav := &ThemedNavBar{}
nav.ExtendBaseWidget(nav)  // Tell BaseWidget: "I am the extended ve
```

**What it does:**

- Links your custom widget to BaseWidget
- Allows Fyne to call your `CreateRenderer()` method
- Enables proper `Refresh()` behavior
- **CRITICAL**: Without it, your widget won't work!

### Interfaces Satisfied by BaseWidget

By embedding `widget.BaseWidget`, you automatically satisfy:

1. **`fyne.Widget` Interface**
    - Requires: `CreateRenderer()` (YOU implement this)
    - Provides: All other widget methods
2. **`fyne.CanvasObject` Interface**

- `Size()`, `Resize()`, `Position()`, `Move()`
- `Visible()`, `Show()`, `Hide()`
- `Refresh()`, `MinSize()`

3. **Optional Interfaces** (if you implement them)
   - `fyne.Tappable` - for click handling
   - `fyne.Hoverable` - for mouse hover (desktop)
   - `fyne.Draggable` - for drag and drop

### Custom Renderer (Advanced)

```
type customRenderer struct {
    card    *CustomCard
    objects []fyne.CanvasObject
}

func (r *customRenderer) Layout(size fyne.Size) {
    // Position objects manually
    r.objects[0].Resize(fyne.NewSize(size.Width, 50))
    r.objects[0].Move(fyne.NewPos(0, 0))
}

func (r *customRenderer) MinSize() fyne.Size {
    // Calculate minimum size
    return fyne.NewSize(200, 100)
}

func (r *customRenderer) Refresh() {
    // Update appearance when data changes
    canvas.Refresh(r.card)
}

func (r *customRenderer) Objects() []fyne.CanvasObject {
    return r.objects
}

func (r *customRenderer) Destroy() {}
```

---

## Mobile Development

### Device Detection

```
// Check if running on mobile
if fyne.CurrentDevice().IsMobile() {
    // Mobile-specific code
}

// Check orientation
if fyne.CurrentDevice().Orientation() == fyne.OrientationHorizontal
    // Landscape
} else {
    // Portrait
}

// Respond to orientation changes
app.Lifecycle().SetOnOrientationChanged(func(orientation fyne.Device
    // Rebuild UI for new orientation
})
```

Touch-Friendly Design

```
// Minimum touch target: 44x44 points (Apple guideline)
btn.SetMinSize(fyne.NewSize(44, 44))

// Use larger fonts on mobile
if fyne.CurrentDevice().IsMobile() {
    text.TextSize = 18
} else {
    text.TextSize = 14
}

// Avoid hover effects (no mouse on mobile)
// Use Tapped instead of MouseIn/MouseOut
```

Mobile Packaging

```
# Android APK
fyne package -os android -appID com.example.app

# iOS (requires macOS + Xcode)
fyne package -os ios -appID com.example.app
```

```
# Release build (optimized, signed)
fyne package -os android -appID com.example.app -release
```

---

## Performance

### Refresh Efficiently

```
// ❌ BAD - Refreshing parent refreshes ALL children
bigContainer.Refresh()  // Expensive!

// ✅ GOOD - Refresh only what changed
changedLabel.Refresh()  // Fast!

// ❌ BAD - Calling Refresh in loop
for i := 0; i < 1000; i++ {
    items[i].Refresh()  // Very slow!
}

// ✅ GOOD - Batch updates, refresh once
for i := 0; i < 1000; i++ {
    items[i].SetText(newText)
}
container.Refresh()  // Single refresh
```

### Resource Management

```
// Bundle resources at compile time
//go:generate fyne bundle -o bundle.go assets/

// Then use bundled resources
icon := canvas.NewImageFromResource(resourceIconPng)
// No file I/O at runtime = faster!

// Remove references to let garbage collector clean up
myContainer.Objects = nil
myWindow.SetContent(nil)

// Clear large images when done
img.Resource = nil
```

---

## Common Patterns

### Navigation Pattern

```
type AppState struct {
    window   fyne.Window
    screens  map[string]fyne.CanvasObject
    history  []string
}

func (as *AppState) ShowScreen(name string) {
    screen := as.screens[name]
    as.window.SetContent(screen)
    as.history = append(as.history, name)
}

func (as *AppState) GoBack() {
    if len(as.history) > 1 {
        as.history = as.history[:len(as.history)-1]
        previous := as.history[len(as.history)-1]
        as.ShowScreen(previous)
    }
}
```

### Clickable Cards Pattern

```
// Content
cardContent := container.NewBorder(nil, nil, icon, price, info)

// Clickable button (invisible)
btn := widget.NewButton("", func() {
    // Handle click
})

// Stack content on button
return container.NewStack(btn, cardContent)
// Button provides click handling
// Content provides appearance
```

### Theme-Aware Colors

```
// Check current theme variant
variant := fyne.CurrentApp().Settings().ThemeVariant()

var bgColor color.Color
if variant == theme.VariantDark {
    bgColor = color.RGBA{0x2A, 0x2A, 0x2A, 0xFF}
} else {
    bgColor = color.RGBA{0xFF, 0xFF, 0xFF, 0xFF}
}
```

---

## Learning Path

### Don't Learn Everything at Once!

Most Fyne developers use 95% pre-built widgets. Focus on what you need.

### Week 1: Just Build

```
// Only learn these:
widget.NewButton, widget.NewLabel, widget.NewEntry
container.NewVBox, container.NewHBox
```

Build something simple with just these!

### Week 2: Add Layout

```
// Learn:
container.NewBorder
container.NewStack
layout.NewSpacer()
```

Now you can make real UIs!

### Week 3: Add Style

```
// Learn:
canvas.NewRectangle (for backgrounds)
theme.PrimaryColor() (for colors)
```

Now it looks professional!

Month 2+: Advanced (Optional)

```
// Only if needed:
Custom widgets
Data binding
Custom renderers
```

Daily Workflow

```
// 1. Start with simple widgets
content := widget.NewLabel("Hello")

// 2. Need custom look? Add canvas
bg := canvas.NewRectangle(color.Blue)
styled := container.NewStack(bg, content)

// 3. Need layout? Add container
layout := container.NewVBox(
    header,
    styled,
    footer,
)

// 4. Need special behavior? Make custom widget
// (But this is rare - maybe 5% of time)
```

---

## Quick Reference

80% of Daily Work

```
// Widgets
widget.NewButton("Text", callback)
widget.NewLabel("Text")
widget.NewEntry()
widget.NewCheck("Text", callback)

// Containers
container.NewVBox(items...)
container.NewHBox(items...)
container.NewStack(items...)
container.NewBorder(top, bottom, left, right, center)
```

```
// That's it!
```

15% of Daily Work

```
// Visual elements
canvas.NewRectangle(color)
canvas.NewText("Text", color)
canvas.NewImageFromResource(res)

// Layout helpers
layout.NewSpacer()
```

5% of Daily Work (Rare)

```
// Custom widgets
// Data binding
// Custom renderers
```

---

## Key Takeaways

1. **App → Window → Content**: Hierarchical structure
2. **Widgets = Interactive**: Buttons, entries, checkboxes
3. **Canvas = Visual**: Text, shapes, images
4. **Containers = Layout**: Automatically arrange children
5. **Theme = Appearance**: Colors, fonts, sizes
6. **Refresh = Update**: Changes need refresh to display
7. **Binding = Reactive**: UI auto-updates with data
8. **Custom Widget = Reusable**: Extend BaseWidget
9. **Mobile = Touch**: Consider touch targets and orientation
10. **Performance = Selective Refresh**: Only refresh what changed

---

## Conclusion

Fyne makes cross-platform GUI development in Go accessible and enjoyable. Focus on:

- Using built-in widgets for 90% of your needs
- Understanding containers for proper layouts
- Creating custom widgets only when necessary
- Following the refresh pattern for updates
- Testing on target platforms early

**Most importantly:** Build your app! You'll learn what you need as you go.

---

## Resources

- Official Documentation: https://developer.fyne.io/
- API Reference: https://pkg.go.dev/fyne.io/fyne/v2
- Examples: https://github.com/fyne-io/examples
- Community: https://github.com/fyne-io/fyne/discussions

---

*This guide covers the essentials of Fyne development. For more advanced topics, refer to the official documentation.*