

THE GADGET FACTORY

Programmer's Reference

VERSION 1.0 / PDF EDITION

23 OCTOBER 2026

A reference built for the workbench

Gadget Factory brings the repeated work of dialog construction, binding, and interaction into a shared Lua framework. This book explains the contracts behind that convenience and the controls you can build with it.

Start with the opening chapters for the framework and its execution model. Use Components and Utilities when you know the control you need. The six worked examples connect the API to complete, independent gadgets.

Finding your way

The contents list chapters and reference topics. PDF bookmarks expand to individual sections; the alphabetical index links directly to methods and concepts. Orange references within the text are clickable and include printed page numbers.

Code you can use

Listings are selectable text. Lua keywords, strings, numbers, and comments use restrained syntax coloring. Long lines wrap visually; a small mark in the left margin identifies a continuation. These wraps are typesetting, not changes to the source.

Each complete example is printed once. All six matching Lua files are also embedded as PDF attachments and supplied in `Examples/Scripts` beside the source documentation. Use the attachments or companion files when exact source formatting matters. Attachment controls vary by PDF reader.

Edition and scope

Version 1.0. Prepared 23 October 2026 from Gadget Factory's source documentation. Forge is the interface designer; GadgetFactory is the shared framework. Installation packaging remains provisional: consult the separate Installation Guide supplied with the eventual distribution.

Native Aspire acceptance is identified where it has been recorded. Simulated checks and browser previews are kept distinct from native application behavior. The Layout Grid example also explains the limits of standard Undo and how to clean up generated work.

Publication design: brushed steel, charcoal, and a small measure of molten orange.

Contents

01 Getting started 1

- Gadget Factory: overview 2
- Quick Start Guide 4

02 Framework & lifecycle 6

- Framework API: overview 7
- Loading Gadget Factory 9
- GadgetDef 10
- Gadget Lifecycle 15
- InitializeScript(job,script_path) 34
- LoadComponents() 36
- InitializeDialog(dlg) 49
- Factory Functions 50

03 Conventions & contracts 58

- Programming Conventions 59
- Framework Contracts 62
- Vectric & Trident Considerations 66

04 HTML primitives 70

- HTML primitives: overview 71
- AddElement() 72
- AddElements() 74
- AddDiv() 80
- AddControlLabel() 86
- Field Primitives 93
- AddLabelSpan() 103
- AddStyle() 108

Contents / continued

05 Components 112

- Components: overview 113
- RadioGroup 120
- CheckboxGroup 137
- ButtonGroup 156
- ToolStrip 173
- Combobox 192
- ListBox 217
- DataPicker 246
- LayerPicker 252
- FileList 257

06 Utilities 277

- Utilities: overview 278
- NumberFormatter 281
- Default Buttons 288
- StatusBar 298

07 Advanced topics 315

- Advanced topics: overview 316
- Event Handling 320
- AddBinding() 329
- AddStateFlag() 340
- GetElementsHtml() 351
- Dynamic HTML 364

08 Worked examples 379

- Using the worked examples 380
- Example 1: A First Dialog 382
- Example 2: A Dimension Calculator 385
- Example 3: Choices and Options 389
- Example 4: Dependent Lists 392
- Example 5: A Preview Workbench 396
- Example 6: The Layout Grid 401

Alphabetical index - 416

CHAPTER 01

01

Getting started

The framework, the designer, and your first gadget.

IN THIS CHAPTER

Gadget Factory: overview 2

Quick Start Guide 4

REFERENCE 1.1

Gadget Factory: overview

Purpose

Gadget Factory is a Lua framework designed to simplify the development of gadgets for Vectric applications.

Its primary purpose is to eliminate the repetitive machinery involved in creating dialogs, managing user interaction, and communicating with Vectric's HTML interface.

By assuming responsibility for these tasks, Gadget Factory allows a gadget's source code to concentrate on its actual purpose rather than the infrastructure required to accomplish it.

The guiding principle is simple: **a gadget should describe what it does, not spend most of its code explaining how to interact with its user.**

What Gadget Factory Provides

Programmatic interface construction. Dialogs are assembled from reusable components, HTML primitives, and optional custom HTML. Gadget Factory generates the necessary HTML automatically, eliminating the need to create and maintain separate dialog files.

Automatic binding and event handling. Interface elements can be connected to Vectric through the framework. Where supported, supplying a callback function is sufficient to establish interaction, without manually implementing the underlying event-dispatch machinery.

Simplified gadget execution. Gadget Factory owns the conventional `main()` entry point and `OnLuaButton_XXXX()` callback dispatcher. Gadget-specific behavior is supplied through clearly defined lifecycle hooks and callbacks.

Unified source code. A gadget's interface definition and application logic can reside in the same Lua source file. Shared libraries and stylesheets remain available without requiring separate HTML documents for individual gadgets.

Separation of presentation and behavior. CSS controls much of the interface's appearance, allowing visual adjustments without restructuring application logic.

Extensibility. Reusable components provide convenient building blocks, but do not prevent developers from using custom HTML, directly invoking framework primitives, or employing Lua's full programming capabilities.

What Gadget Factory Is Not

Gadget Factory is not a replacement for Lua, Vectric's API, or the programmer's judgment. It does not attempt to determine what a gadget should accomplish or how its application-specific operations should be implemented.

It is not a visual programming environment. Tools such as Forge can make interface construction more convenient, but are separate from the framework and are not required to use it.

Nor is Gadget Factory intended to impose a restrictive programming model. Lua's flexibility - including tables, functions as values, dynamic typing, and custom control structures - remains available to the gadget author.

The framework provides conveniences and conventions, not artificial limitations.

Design Philosophy

Gadget Factory seeks to centralize repetitive complexity without sacrificing flexibility.

Tasks common to many gadgets belong in the framework, where they can be implemented, tested, and maintained once. Functionality unique to an individual gadget belongs in that gadget's source.

The framework should remain understandable and predictable. Its public interfaces, conventions, limitations, and significant implementation behaviors are documented in this Programmer's Reference.

Gadget Factory exists to make programming gadgets easier, not to place boundaries on what a programmer can create.

REFERENCE 1.2

Quick Start Guide

From an installed framework to your first gadget in a few simple steps.

This guide begins after **Forge** and the shared **GadgetFactory** framework have been installed for your Vectric application.

> **Installation:** The distribution package is still being finalized for this edition. For package contents, supported application versions, deployment locations, and installation steps, refer to the separate **Installation Guide** supplied with the eventual distribution. This Quick Start does not prescribe archive names, installer filenames, or a manual extraction procedure.

1. Open Forge

Start your Vectric application and launch **Forge** from the Gadgets menu, following any startup instructions in the Installation Guide.

Forge is the interface designer. GadgetFactory is the shared framework used by the gadgets you create.

2. Design Your Interface

If you've ever used a visual form or dialog editor, you'll feel right at home.

- **Left:** Select a field or component to add to your dialog.
- **Right:** Edit its properties, then click **ADD**.
- Repeat until your interface looks the way you want.

3. Save Your Design

If your design is worth keeping, **SAVE it!**

Saving preserves the editable design so you can return to it later. Keep that design when you want to reopen it in Forge and continue editing.

4. Export Your Gadget

Use **EXPORT** to generate a Lua gadget script. The exported file includes your interface and the function stubs needed to add your gadget's actual functionality.

The runnable gadget is a `.lua` file. Save it in the gadget location appropriate to your Vectric application, as described in the Installation Guide. **SAVE** preserves the editable design; **EXPORT** produces the script you run and extend. Keep both when you need both workflows.

5. Run Your Gadget

Vectric normally discovers its gadgets when the application starts. If your exported script introduces a new gadget filename, restart Vectric to refresh its Gadgets menu.

Find your new gadget in the menu and run it!

You now have a GadgetFactory-powered gadget, ready for you to add whatever functionality you have in mind.

Understand the Generated Script

Before extending the script, review Loading Gadget Factory (p. 9) and GadgetDef (p. 10). GadgetFactory supplies the global `main()` entry point; your gadget supplies the documented lifecycle hooks. Build the interface in `InitializeScript(job, script_path)` and perform the final operation in `ProcessResults()` after acceptance.

The framework entry file documented in this reference is `GadgetFactory.gfm`. It is a shared runtime module, not a separate gadget to launch from the menu. The documented loader locates it relative to the `Vectric Files\Gadgets` root; see Loading Gadget Factory (p. 9) for the exact code and its directory assumptions. Use the Installation Guide to establish the supported installation layout rather than inventing another path here.

Load component libraries through LoadComponents() (p. 36) using library names such as `"RadioGroup"`, without a path or extension. The documented loader prefers a compiled `.gfc` module when present, otherwise a source `.gfm` module. A present but invalid `.gfc` does not automatically fall back to `.gfm`. When combining exported snippets, include their required library names in the gadget's `LoadComponents()` call.

A Handy Time-Saver

Once your installation and gadget location are established, you can avoid repeatedly restarting Vectric during development.

Before launching Vectric, create an empty file in that gadget location, such as `MyGadget.lua`. Vectric will discover the filename and include it in the menu. Later, overwrite this placeholder with your exported gadget.

You can then launch the updated gadget without restarting Vectric.

Don't try to run the empty placeholder before replacing it with actual Lua code.

Continue with Loading Gadget Factory (p. 9) and GadgetDef (p. 10) to understand how the framework and your gadget work together.

CHAPTER 02

02

Framework & lifecycle

Loading, execution, and the contract between your gadget and the factory.

IN THIS CHAPTER

Framework API: overview 7

Loading Gadget Factory 9

GadgetDef 10

Gadget Lifecycle 15

InitializeScript(job,script_path) 34

LoadComponents() 36

InitializeDialog(dlg) 49

Factory Functions 50

REFERENCE 2.1

Framework API: overview

The **Framework API** provides the foundation for developing Vectric gadgets with Gadget Factory.

It manages the gadget's execution lifecycle, including job validation, HTML document generation, native dialog creation, field bindings, event dispatching, result processing, and cleanup.

Rather than implementing these operations independently for every gadget, the programmer loads Gadget Factory and supplies a definition describing the application's requirements and behavior.

Gadget Factory then coordinates the execution of the gadget through a series of programmer-defined callbacks.

This section explains how those pieces fit together and provides the primary reference for the framework's public interface.

Recommended reading order

The following topics are arranged in a logical learning sequence rather than alphabetically.

1. **Loading Gadget Factory** (p. 9) - Establish the framework in a gadget script, locate its installation directory, and obtain the public `GF` interface.
2. **GadgetDef** (p. 10) - Configure the gadget's requirements, dialog dimensions, title, and lifecycle callbacks through `GF.GadgetDef`.
3. **Gadget Lifecycle** (p. 15) - Understand the complete execution sequence, from Vectric's initial call to `main()` through dialog construction, user interaction, result processing, and cleanup.
4. **InitializeScript(job,script_path)** (p. 34) - Prepare application state, examine the current job, construct the HTML interface, register callbacks, and configure the components needed by the gadget.
5. **LoadComponents()** (p. 36) - Load reusable component libraries, understand source and compiled module precedence, and make their public APIs and stylesheets available.
6. **InitializeDialog(dlg)** (p. 49) - Perform operations requiring the instantiated Vectric dialog after its native field bindings have been established.
7. **Factory Functions** (p. 50) - Consult the framework's public function reference when looking for a particular capability or navigating to its detailed documentation.

The chapters may also be used independently as a reference once the programmer is familiar with Gadget Factory's execution model.

Understanding the framework's responsibilities

Gadget Factory establishes the global `main()` entry point expected by Vectric.

During execution, it obtains the required job, invokes the gadget's initialization functions, constructs the dialog, registers native field bindings, and displays the interface.

Once the user completes the dialog, Gadget Factory invokes the appropriate processing callback and performs its normal cleanup.

The programmer is responsible for supplying the application's data, business logic, interface definitions, and responses to user interaction.

This division of responsibilities allows even sophisticated gadgets to use a relatively small amount of framework-specific initialization code.

Construction versus execution

An important distinction runs throughout the Framework API.

During interface construction, the programmer defines what the dialog will contain and registers the functionality it will require.

During dialog execution, the programmer interacts with the instantiated controls, retrieves user input, updates displayed information, and responds to events.

For example, InitializeScript (p. 34) creates the interface before `HTML_Dialog` exists, while InitializeDialog (p. 49) receives the native dialog after field registration.

Understanding this separation helps prevent operations from being attempted before their required facilities are available.

The Gadget Lifecycle (p. 15) chapter explains these stages in detail.

Related sections

The Framework API establishes the machinery used throughout Gadget Factory. The following sections explain how to put that machinery to work.

Section	Purpose
HTML Primitives (p. 71)	Construct interfaces using HTML, CSS, and Gadget Factory's field helpers.
Components (p. 113)	Create reusable groups, lists, and other interface controls.
Utilities (p. 278)	Use numeric formatting, dialog buttons, and status feedback.
Advanced Topics (p. 316)	Extend standard behavior through custom bindings, event handling, and dynamic HTML.

Together, these sections provide the tools needed to construct and customize Vectric gadget interfaces without repeatedly implementing the underlying dialog machinery.

Summary

The Framework API defines the relationship between a gadget and Gadget Factory.

The programmer configures the application, defines its interface, and supplies the functions that implement its behavior.

Gadget Factory manages the surrounding infrastructure and invokes those functions at the appropriate points in the gadget's lifecycle.

The programmer defines what the gadget should accomplish. Gadget Factory provides the machinery to make it happen.

REFERENCE 2.2

Loading Gadget Factory

Before a gadget can use Gadget Factory, it must load the framework.

Unlike a conventional Vectric gadget, a Gadget Factory script does not implement its own `main()` function. Gadget Factory provides this entry point and manages the sequence of initialization, dialog creation, user interaction, and cleanup.

For this to work, Gadget Factory must be loaded before anything that depends on it.

Getting Started

Place these two statements at the beginning of every gadget script that uses Gadget Factory:

```lua
local root = assert(
    debug.getinfo(1, "S").source:sub(2):gsub("/", "\\")
        :match("^(.-\\Vectric Files\\Gadgets\\)"),
    "Cannot locate Vectric Files\\Gadgets root.")
local GF = assert(loadfile(root .. "GadgetFactory.gfm"),
    "Cannot load GadgetFactory.")(root)
```

These instructions locate the Gadget Factory framework, load it, and make its functions available through the `GF` variable.

They also provide the Factory with the location of its supporting files, including component libraries and stylesheets.

Simply copy these lines into your script and leave them alone. There's no need to understand their internal workings to use the framework.

Where Gadget Factory Lives

The framework must be installed at:

`...\Vectric Files\Gadgets\GadgetFactory.gfm`

This location serves several purposes:

- It provides a common location accessible to all gadgets using the framework.
- It keeps the framework separate from individual gadget folders and out of Vectric's runnable gadget menu.
- It establishes a predictable location for finding the component libraries, stylesheets, and other supporting resources.

The loading instructions depend on this folder arrangement. Moving Gadget Factory elsewhere requires corresponding changes to how it is located.

What Happens Next?

Once Gadget Factory has been loaded, the script can begin defining its gadget.

The next step is to supply `GF.GadgetDef`, which describes the gadget's basic requirements and identifies the functions Gadget Factory will call during execution.

Continue with Defining a Gadget (p. 10).

REFERENCE 2.3

GadgetDef

Introduction

`GadgetDef` is the starting point for creating a gadget with Gadget Factory. It tells the framework what the gadget needs and provides the functions that will be called at different stages of its execution.

In a conventional Vectric gadget, the programmer implements `main()` to manage initialization, create and display a dialog, handle the results, and perform any necessary cleanup.

Gadget Factory handles this sequence automatically.

Instead of implementing `main()`, the programmer supplies a table named `GF.GadgetDef`, containing the gadget's basic properties and references to functions called *hooks*.

A hook is simply a Lua function that Gadget Factory calls at a particular point during execution. The programmer decides what the function does; Gadget Factory decides when to call it.

This keeps the gadget's code focused on its intended purpose, while the framework handles the repetitive work of managing its execution.

Defining GadgetDef

A typical definition looks like this:

```lua
GF.GadgetDef = {
    Title = "My Gadget",
    Width = 900,
    Height = 600,
    Job = true,

    InitializeScript = InitializeScript,
    InitializeDialog = InitializeDialog,
    ProcessResults = ProcessResults,
    DiscardResults = DiscardResults,
    FinalizeScript = FinalizeScript,
    OnError = OnError,
}
```

The example assumes that `GF` is available and the named functions have already been defined.

Only `InitializeScript` is mandatory. Other hooks can be omitted when they aren't needed.

`GadgetDef` has two kinds of entries:

- **Properties** describe the gadget's requirements and appearance.
- **Hooks** identify functions the framework will call during execution.

The values are ordinary Lua values. Strings, variables, expressions, and function references can be used wherever appropriate.

Gadget Properties

Property	Description
`Title`	Text displayed in the dialog's title bar.
`Width`	Initial dialog width.
`Height`	Initial dialog height.
`Background`	Optional CSS `background-color` value. Gadget Factory appends a body rule with `!important`.
`Job`	Boolean indicating whether an existing Vectric job is required.

See Stylesheet integration (p. 110) for the background rule's precedence.

The Job Requirement

`GadgetDef.Job` is a Boolean, **not a Vectric job object**.

When `Job = true`, Gadget Factory checks that a Vectric job is open before allowing initialization to proceed.

When `Job = false`, or the property is omitted, this requirement is disabled. Initialization proceeds whether a job is open or not.

This does not prevent the gadget from using a job. It simply leaves responsibility for determining its job requirements to the programmer.

For example, a gadget that needs to open a particular job, create a new one, or operate without any job can set `Job = false` and handle those requirements in `InitializeScript()`.

Gadget Hooks

Hooks allow the programmer to supply code without taking responsibility for the execution machinery.

Hook	Required	Purpose
`InitializeScript`	Yes	Prepares the gadget before dialog creation.
`InitializeDialog`	No	Provides access to the created dialog object.
`ProcessResults`	No	Performs the gadget's work after Execute.
`DiscardResults`	No	Handles cancellation when special action is needed.

Hook	Required	Purpose
`FinalizeScript`	No	Performs final housekeeping before the framework shuts down.
`OnError`	No	Provides custom error reporting for framework-dispatched callbacks.

InitializeScript(job, script_path)

This is the **only required hook**.

It is called before Gadget Factory creates the dialog and is the appropriate place to load components, construct the interface, initialize variables, and prepare anything needed for execution.

Gadget Factory supplies two arguments:

- `job` - a validated Vectric job object when `Job` is enabled; otherwise the configured `Job` value, normally `false` or `nil`.
- `script_path` - the directory containing the gadget script, useful for locating associated files or other resources.

The `Job` property determines whether Gadget Factory obtains a job object before calling this hook. Disabling the requirement does not supply a job object.

Returning `false` from `InitializeScript()` aborts initialization and prevents the dialog from appearing. A missing `InitializeScript()` is a programming error.

If initialization has opened files or acquired other resources before returning `false`, the initializer is responsible for cleaning them up.

InitializeDialog(dlg)

Called after Gadget Factory creates and binds the dialog, but before displaying it.

The `dlg` argument provides access to Vectric's `HTML_Dialog` object for operations that require direct interaction with the dialog.

The same object is also available through `GF.Dialog` after creation.

Most gadgets will not need this hook.

ProcessResults()

Called when the user completes the dialog using Execute.

This is where the gadget normally performs its intended work, using the selections and values established through the interface.

Gadget Factory handles the dialog interaction; the programmer handles the actual operation.

DiscardResults()

Called when the dialog is canceled.

Most gadgets have nothing to do when the user presses Cancel, so this hook can be omitted.

It is useful when cancellation requires special handling, such as restoring temporary changes or releasing resources that should not persist.

FinalizeScript()

Called after the dialog's results have been processed or discarded, immediately before Gadget Factory's internal `Done()` routine performs the framework's own cleanup.

It receives no arguments. The gadget's own functions and variables are already available within its Lua context.

This hook is intended for normal completion, not emergency error recovery.

It is not automatically called when initialization is aborted or an unhandled Lua error interrupts execution.

OnError(err)

Provides a way to customize the reporting of errors caught by Gadget Factory's callback mechanism.

It does not replace Lua's normal error handling and should not be assumed to catch every error that may occur during execution.

The `err` argument is the intercepted Lua error value. The hook's return value is ignored.

See Error handling (p. 327) for callback behavior and the limits of this protection.

The Execution Sequence

Once `GadgetDef` is established, Gadget Factory controls the execution sequence:

1. Check whether an existing job is required.
2. Call `InitializeScript()` to prepare the gadget.
3. Construct the dialog and bind its controls.
4. Call `InitializeDialog()` if provided.
5. Display the dialog and handle user interaction.
6. Call `ProcessResults()` or `DiscardResults()`, depending on how the dialog closes.
7. Call `FinalizeScript()` if provided.
8. Call the internal `Done()` routine to perform framework cleanup.

If initialization is aborted, the remaining steps are skipped.

A Minimal Definition

Not every gadget needs every hook. A simple gadget might require only:

```lua
GF.GadgetDef = {
    Title = "My Gadget",
    Width = 600,
    Height = 400,
    Job = false,
    InitializeScript = InitializeScript,
    ProcessResults = ProcessResults,
}
```

The programmer provides the two referenced functions, and Gadget Factory manages the rest.

As the gadget becomes more sophisticated, additional hooks can be included without changing the overall structure.

Keeping Control

Gadget Factory simplifies the normal execution of a Vectric gadget, but does not dictate its behavior.

The programmer remains free to use Lua functions, tables, variables, and Vectric's API as needed. Hooks are ordinary Lua functions, not a separate programming language or a restrictive set of commands.

GadgetDef provides the structure. The programmer provides the purpose.

REFERENCE 2.4

Gadget Lifecycle

Gadget Factory manages the complete execution lifecycle of a Vectric gadget, from validating the current job and constructing the interface to processing dialog results and performing cleanup.

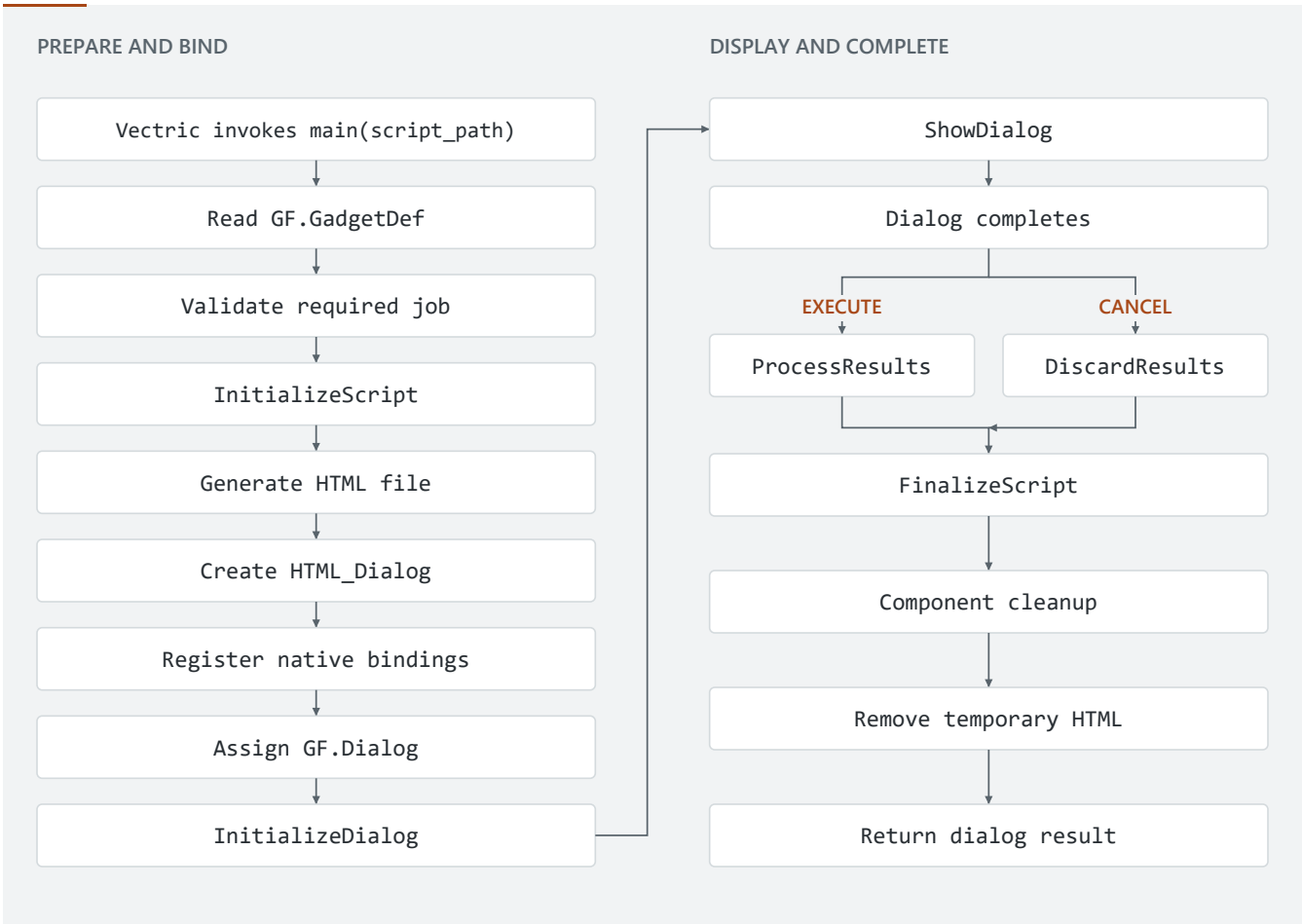
Rather than requiring every gadget to implement its own `main()` function, create an `HTML_Dialog`, register fields, and manage the dialog's completion, Gadget Factory centralizes those operations.

The gadget programmer supplies a definition table containing configuration properties and callback functions. Gadget Factory invokes those functions at the appropriate stages of execution.

This separation allows the programmer to concentrate on **what the gadget does**, while Gadget Factory manages **when and how its interface operates**.

The lifecycle at a glance

A normal Gadget Factory execution follows this sequence:



Several stages are conditional. For example, `InitializeDialog()`, `ProcessResults()`, `DiscardResults()`, and `FinalizeScript()` execute only when the gadget supplies the corresponding callback.

The diagram represents normal execution. Early termination and unhandled errors are discussed later.

GadgetDef

The lifecycle is configured through `GF.GadgetDef`.

Basic structure

```lua
GF.GadgetDef = {
    Job    = true,
    Width  = 500,
    Height = 350,
    Title  = "My Gadget",

    InitializeScript = InitializeScript,
    InitializeDialog = InitializeDialog,
    ProcessResults   = ProcessResults,
    DiscardResults   = DiscardResults,
    FinalizeScript   = FinalizeScript,
}
```

The table identifies the functions Gadget Factory should invoke and supplies the information needed to construct the dialog.

Lifecycle properties

Property	Purpose
`Job`	Require an existing Vectric job when true.
`Width`	Dialog width.
`Height`	Dialog height.
`Title`	Dialog window title.
`InitializeScript`	Prepare application state and construct the interface.
`InitializeDialog`	Perform optional adjustments after creating and binding the dialog.
`ProcessResults`	Process successful dialog completion.
`DiscardResults`	Respond to dialog cancellation.
`FinalizeScript`	Perform optional final work after normal dialog completion.
`OnError`	Optional reporter for errors intercepted by Gadget Factory's event dispatcher.

Gadget Factory also recognizes an optional `Background` property, which supplies a dialog background-color override.

The individual configuration properties are described in GadgetDef (p. 10).

Required and optional callbacks

`InitializeScript` is required by the current factory implementation.

Gadget Factory calls it unconditionally:

```lua
Def.InitializeScript(job, script_path)
```

The remaining lifecycle callbacks are optional.

If one is absent, Gadget Factory skips the corresponding operation.

A gadget therefore needs only the callbacks appropriate to its purpose.

Stage 1: The Vectric entry point

Vectric begins gadget execution by invoking the global Lua function:

```lua
main(script_path)
```

In a Gadget Factory application, this function is supplied by the framework.

The gadget programmer does not normally define a separate `main()` function.

When the gadget loads `GadgetFactory.gfm`, the framework establishes the global entry point and returns its public `GF` table.

The programmer then configures `GF.GadgetDef` before Vectric invokes the entry point.

The script_path argument

`script_path` identifies the gadget's script directory.

Gadget Factory uses this location when creating its temporary HTML document.

The same value is forwarded to `InitializeScript()` so the gadget can locate resources relative to its own directory.

This can be useful when loading templates, configuration files, or other assets distributed with the gadget.

Stage 2: Job validation

Before invoking `InitializeScript()`, Gadget Factory examines the `Job` property.

```lua
Job = true
```

indicates that the gadget requires an existing Vectric job.

The framework attempts to obtain the current job:

```lua
job = VectricJob()
```

It then checks:

```lua
if not job.Exists then
    DisplayMessageBox("Open a job and try again.")
    return false
end
```

If no job exists, execution terminates before the gadget's initialization callback is invoked.

Gadgets that do not require a job

For utilities that can operate independently of an open Vectric job:

```lua
Job = false
```

Gadget Factory does not construct or validate a `VectricJob` object.

The `job` parameter passed to `InitializeScript()` is then the configured value, normally `false` or `nil`.

Such gadgets may perform tasks that do not depend on job geometry, toolpaths, or other job-specific information.

An important distinction

The `Job` property controls whether Gadget Factory requires and supplies an existing job object.

It does not determine whether a gadget can subsequently create or access Vectric job objects through its own code.

Those operations remain the programmer's responsibility.

Stage 3: InitializeScript

`InitializeScript()` is the principal preparation and interface-construction callback.

Syntax

```lua
local function InitializeScript(job, script_path)
    -- Prepare the gadget
end
```

Parameters

Parameter	Description
`job`	Current Vectric job when requested and validated. Otherwise, the configured Job value.
`script_path`	Directory containing the gadget's entry-point script.

The callback executes **before the native `HTML_Dialog` object is constructed**.

This is where the programmer normally performs tasks such as:

- Establishing initial application values.
- Examining the current Vectric job or selection.
- Loading component libraries.
- Registering component instances.
- Adding CSS styles.
- Constructing HTML through Gadget Factory primitives.
- Registering bindings and event handlers.
- Configuring utilities such as StatusBar.
- Suppressing the default buttons when required.

Example: Sectio Divina

Sectio Divina begins initialization by retaining the job reference and examining the current selection:

```lua
local function InitializeScript(job)
    Job = job

    local selection = Job.Selection

    if selection.IsEmpty then
        X = Job.XLength / 2
        Y = X / PHI
    else
        local bounds = selection:GetBoundingBox()
        X = bounds.XLength
        Y = bounds.YLength
    end

    -- Additional initialization
end
```

It subsequently loads its component libraries:

```lua
GF.LoadComponents {"RadioGroup", "ButtonGroup"}
```

and constructs the dialog using `GF.AddElements()`, `GF.AddStyle()`, and its component APIs.

All this occurs before the native dialog exists.

Aborting initialization

`InitializeScript()` may terminate gadget execution by returning exactly `false`.

```lua
local function InitializeScript(job)
    if not SomeRequiredCondition then
        DisplayMessageBox("Required conditions are not met.")
        return false
    end

    -- Continue initialization
end
```

The factory checks:

```lua
if Def.InitializeScript(job, script_path) == false then
    return false
end
```

Notice the explicit comparison with `false`.

Only an actual Boolean `false` aborts initialization.

Returning `nil`, returning `true`, or reaching the end of the function without an explicit return allows execution to continue.

What happens when initialization is aborted?

If `InitializeScript()` returns `false`:

- The native dialog is not created.
- No collected native bindings are applied.
- `InitializeDialog()` is not called.
- `ShowDialog()` is not called.
- No result-processing callback is invoked.
- `FinalizeScript()` and Gadget Factory's normal cleanup routine are not invoked.

This is an early exit from the framework's `main()` function.

See InitializeScript (p. 34) for additional details about constructing the interface.

Stage 4: HTML document generation

After `InitializeScript()` completes successfully, Gadget Factory creates the HTML document used by Vectric.

The temporary file is named:

```text
GadgetTemp.html
```

and is placed in the gadget's script directory.

During document generation, Gadget Factory combines:

1. The document header and metadata.
2. Links to registered CSS stylesheets.
3. CSS supplied through AddStyle (p. 108).
4. HTML collected through primitives and components.
5. The standard Execute and Cancel buttons, unless suppressed.
6. The closing document tags.

The result is a complete HTML document that Vectric can load into its embedded browser.

A useful distinction

Functions such as `GF.AddElement()` and `GF.AddElements()` do not normally write directly to this temporary file.

They populate Gadget Factory's internal content collection.

The document is assembled later, after `InitializeScript()` has finished preparing its content.

This allows HTML construction to occur throughout initialization without repeatedly rebuilding the document.

Stage 5: HTML_Dialog creation

After generating the HTML file, Gadget Factory constructs Vectric's native dialog:

```lua
Dlg = HTML_Dialog(
    false,
    DOM(),
    Def.Width,
    Def.Height,
    Def.Title
)
```

`DOM()` creates the temporary HTML file and returns its location.

The resulting `Dlg` object provides the native methods used for field registration, value retrieval, updates, and dialog display.

At this point, the Lua dialog object exists, but its HTML document has not necessarily loaded into the embedded browser.

That distinction becomes important when using dynamic HTML.

Stage 6: Native field and component binding

Once the dialog exists, Gadget Factory executes the bindings collected during interface initialization.

The relevant implementation is:

```lua
for _, binding in ipairs(bindings) do
    if type(binding) == "function" then
        binding(Dlg)
    else
        binding:Bind(Dlg)
    end
end
```

There are two supported binding forms.

Function bindings

Bindings registered through AddBinding (p. 329) receive the native dialog object.

For example:

```lua
GF.AddBinding(function(dlg)
    dlg:AddCheckBox("chkReady", false)
end)
```

When the binding phase begins, Gadget Factory invokes the function with the newly constructed dialog.

Component bindings

Reusable components register themselves with Gadget Factory during construction.

For components that require native fields, the framework calls their `Bind()` methods.

For example, RadioGroup uses:

```lua
function RadioGroup:Bind(dlg)
    self.dlg = dlg
    dlg:AddRadioGroup(self.id, self.selected or 1)
end
```

This registers the component's native radio-group field and retains the dialog reference for subsequent operations.

Binding order

Bindings execute in the order in which they were registered.

This includes both programmer-supplied binding functions and component objects.

The binding phase occurs before `GF.Dialog` is assigned.

Consequently, binding functions should use their supplied `dlg` parameter rather than relying on `GF.Dialog`.

Stage 7: GF.Dialog and InitializeDialog

After every registered binding has executed, Gadget Factory exposes the dialog through:

```lua
GF.Dialog = Dlg
```

This makes the active dialog available to gadget functions that do not maintain their own dialog reference.

The framework then invokes the optional `InitializeDialog()` callback.

Syntax

```lua
local function InitializeDialog(dialog)
    -- Native dialog adjustments
end
```

Parameter

Parameter	Description
`dialog`	The instantiated and bound Vectric `HTML_Dialog` object.

Example: Sectio Divina

Sectio Divina retains the dialog reference in a script-level variable:

```lua
local function InitializeDialog(dialog)
    Dlg = dialog
end
```

This allows the gadget's other functions to access Vectric's native field methods through `Dlg`.

Alternatively, callbacks may use `GF.Dialog`.

InitializeDialog versus InitializeScript

These callbacks execute at different stages.

InitializeScript	InitializeDialog
Before HTML_Dialog construction	After HTML_Dialog construction
Prepares application state	Receives the native dialog
Generates interface content	Works with the instantiated dialog
Registers bindings	Executes after bindings have been applied
Required	Optional

Document availability

Native Aspire verification found that `GF.ExecutionEnabled(true)` called from `InitializeDialog()` failed to enable the Execute button. The function relies on `SetOuterHtml()`.

At this stage, the native dialog and its field bindings exist. The result establishes that DOM-dependent updates cannot be relied upon here; it does not conclusively establish that the DOM is absent.

Therefore, operations such as:

```lua
dialog:SetInnerHtml("details", "<b>Ready</b>")
```

should not be assumed to work reliably from `InitializeDialog()`.

For DOM-dependent updates, use an appropriate callback after the HTML document has loaded.

See The native findings (p. 366) and Post-load initialization (p. 367) for the evidence and the practical timing rule.

Stage 8: ShowDialog

After initialization, Gadget Factory displays the dialog:

```lua
local ok = Dlg:ShowDialog()
```

`ShowDialog()` transfers interaction to Vectric's native dialog system and does not return until the dialog completes.

While the dialog is open, the user can interact with its controls.

During dialog execution

Depending on the gadget, user interactions may cause:

- Gadget Factory event handlers to execute.
- Component selection callbacks to run.
- Native field values to change.

- StatusBar messages to update.
- HTML content to be replaced dynamically.
- Standard Execute availability to change.

These interactions are handled through the mechanisms established during initialization.

Gadget Factory's event dispatcher (p. 320) routes LuaButton events to registered functions.

The dispatcher is independent of the main lifecycle's result-processing callbacks.

Event callbacks do not normally complete the dialog

An ordinary event handler can update fields or perform application-specific work while the dialog remains visible.

For example:

```lua
GF.AddHandler("btnPreview", RefreshPreview)
```

The registered callback executes when its corresponding LuaButton is activated.

It does not automatically cause `ShowDialog()` to return.

The dialog completes through Vectric's native completion mechanism, normally using controls such as `ButtonOK` and `ButtonCancel`.

See Default Buttons (p. 288).

Stage 9: Processing the dialog result

When `ShowDialog()` returns, Gadget Factory examines its result.

The relevant implementation is:

```lua
if ok then
    if Def.ProcessResults then
        Def.ProcessResults()
    end
elseif Def.DiscardResults then
    Def.DiscardResults()
end
```

ProcessResults

`ProcessResults()` is invoked when `ShowDialog()` reports successful completion.

```lua
local function ProcessResults()
    -- Apply the gadget's accepted results
end
```

Parameters: None supplied by Gadget Factory.

Return value: Ignored by Gadget Factory.

Typical responsibilities include:

- Retrieving the final values of dialog controls.
- Reading component selections.
- Validating or applying the accepted settings.
- Creating or modifying Vectric objects.
- Generating toolpaths.
- Performing other application-specific operations.

For example:

```lua
local function ProcessResults()
    local depth = Dlg:GetDoubleField("txtDepth")

    -- Use the accepted depth
end
```

The dialog's bound fields have already been registered, and the application can use the native dialog reference established earlier.

DiscardResults

`DiscardResults()` is invoked when `ShowDialog()` returns a false or otherwise falsey result.

```lua
local function DiscardResults()
    -- Respond to cancellation
end
```

Parameters: None supplied by Gadget Factory.

Return value: Ignored by Gadget Factory.

A gadget may use this callback to discard temporary application state or perform cancellation-specific operations.

Many gadgets need no explicit cancellation processing.

In those cases, `DiscardResults()` can be omitted.

An important distinction

Gadget Factory does not infer success or failure from the return value of `ProcessResults()`.

The dialog's original `ShowDialog()` result determines which branch is executed and ultimately determines the Boolean returned by the framework's `main()`.

Consequently:

```lua
local function ProcessResults()
    return false
end
```

does not convert a successfully completed dialog into a failed `main()` result.

If the gadget requires additional success or failure reporting, that responsibility must be handled explicitly by its application logic.

Stage 10: FinalizeScript

`FinalizeScript()` is an optional callback invoked after normal dialog completion and result processing.

Syntax

```lua
local function FinalizeScript()
    -- Final operations
end
```

Parameters: None supplied by Gadget Factory.

Return value: Ignored.

The callback executes whether the dialog completed successfully or was canceled, provided normal control flow reaches this stage.

It executes after `ProcessResults()` or `DiscardResults()`, and before Gadget Factory's internal cleanup.

Typical uses

A gadget might use `FinalizeScript()` to:

- Release application-owned resources.
- Restore temporary settings.
- Perform final application housekeeping.
- Consolidate work that should happen after either result branch.

What FinalizeScript does not guarantee

Despite its name, `FinalizeScript()` is **not equivalent to a protected `finally` block**.

Gadget Factory calls it through an ordinary conditional function invocation:

```lua
if Def.FinalizeScript then
    Def.FinalizeScript()
end
```

It does not execute if the gadget terminates earlier because:

- The required job is unavailable.
- `InitializeScript()` explicitly returns `false`.
- An earlier unprotected operation raises an error and prevents execution from reaching finalization.

This distinction matters when designing cleanup for resources that must be released even if execution fails unexpectedly.

Stage 11: Gadget Factory cleanup

After `FinalizeScript()`, Gadget Factory invokes its private `Done()` function.

The cleanup routine performs two principal operations.

Component cleanup

Gadget Factory traverses its binding collection and invokes `Done()` on registered component objects that provide such a method.

The essential implementation is:

```lua
for _, binding in ipairs(bindings) do
    if type(binding) == "table" and binding.Done then
        binding:Done()
    end
end
```

Components may use this opportunity to release dialog references, retain their final selection, or complete other internal operations.

For example, FileList uses its cleanup mechanism to persist committed details changes.

Ordinary binding functions do not receive a corresponding cleanup call.

Temporary HTML removal

Gadget Factory then removes the generated HTML file:

```lua
if DOMfile then
    os.remove(DOMfile)
    DOMfile = nil
end
```

This removes `GadgetTemp.html`, which was created solely to provide the dialog's generated HTML document.

The programmer does not normally need to manage this file.

Cleanup ordering

The order is worth noting:

```
ProcessResults or DiscardResults
              |
              ▼
       FinalizeScript
              |
              ▼
      Component Done()
              |
              ▼
   Remove GadgetTemp.html
```

Component cleanup occurs **after** the gadget's result-processing and finalization callbacks.

This means components can still hold application state during those callbacks.

The framework manages its own cleanup; the programmer should not invoke its private `Done()` function.

Stage 12: Returning to Vectric

After cleanup, Gadget Factory returns the Boolean interpretation of the dialog result:

```lua
return not not ok
```

The double negation is a standard Lua idiom for converting a value into an actual Boolean.

If `ShowDialog()` returned a truthy value, `main()` returns `true`.

Otherwise, it returns `false`.

The return values of `ProcessResults()`, `DiscardResults()`, and `FinalizeScript()` do not alter this result.

This concludes normal gadget execution.

A complete lifecycle example

The following example illustrates all five principal lifecycle callbacks.

It assumes that `GF` has already been loaded as described in the Loading Gadget Factory (p. 9).

```lua
local Dlg

local function InitializeScript(job, script_path)

    GF.AddTextField("txtName", "Untitled")

    -- Supply custom completion buttons
    GF.NoDefaultButtons()

    GF.AddElement([[
        <input type="button" id="ButtonOK"
               class="LuaButton" value="Accept">

        <input type="button" id="ButtonCancel"
               class="LuaButton" value="Cancel">
    ]])
end

local function InitializeDialog(dialog)
    Dlg = dialog
end

local function ProcessResults()
    local name = Dlg:GetTextField("txtName")

    -- Process the accepted name
end

local function DiscardResults()
    -- Optional cancellation handling
end

local function FinalizeScript()
    -- Optional final housekeeping
end

GF.GadgetDef = {
    Job    = false,
    Width  = 400,
    Height = 250,
    Title  = "Lifecycle Example",

    InitializeScript = InitializeScript,
    InitializeDialog = InitializeDialog,
    ProcessResults   = ProcessResults,
    DiscardResults   = DiscardResults,
    FinalizeScript   = FinalizeScript,
}
```

The custom completion buttons are included to keep the example independent of Gadget Factory's initially inactive standard Execute button.

For an ordinary gadget, the standard buttons and ExecutionEnabled (p. 289) may be more convenient.

Execution sequence

With this configuration:

1. Gadget Factory skips job validation because `Job` is `false`.
2. `InitializeScript()` creates the TextField and custom buttons.
3. The framework generates the HTML document.
4. `HTML_Dialog` is constructed and the TextField registered.
5. `InitializeDialog()` retains the native dialog reference.
6. `ShowDialog()` displays the interface.
7. The user accepts or cancels.
8. The appropriate result callback executes.
9. `FinalizeScript()` runs.
10. Gadget Factory performs component cleanup and removes its temporary HTML.
11. The framework returns the dialog-completion result.

The programmer does not need to implement `main()` or manually coordinate these stages.

Error handling and lifecycle boundaries

Gadget Factory's callback dispatcher uses protected Lua calls to intercept errors from ordinary registered event handlers.

Those errors are reported through:

```lua
GF.OnError(err)
```

The programmer may provide a custom error reporter using `GF.GadgetDef.OnError`.

However, **this protection does not extend automatically to every lifecycle callback**.

Operations such as:

```lua
InitializeScript()
InitializeDialog()
ProcessResults()
FinalizeScript()
```

are invoked through normal Lua function calls.

Likewise, the native binding loop does not wrap every binding in a protected call.

An error in one of these operations can interrupt the lifecycle before later callbacks or cleanup execute.

See Event Handling (p. 320) for the protected LuaButton dispatcher and GadgetDef (p. 10) for configuring the error reporter.

For resources requiring guaranteed cleanup under failure conditions, the gadget must implement appropriate error-handling arrangements of its own.

Important considerations

- **Factory-owned main:** Gadget Factory defines the global `main()` entry point. Gadget programmers normally do not define another.
- **Definition required:** `GF.GadgetDef` must be configured before Vectric invokes `main()`.
- **InitializeScript required:** The current implementation invokes it unconditionally.
- **Job validation:** When enabled, the required job is checked before initialization.
- **Explicit abort:** Only `InitializeScript()` returning exactly `false` prevents dialog construction.
- **HTML construction timing:** HTML and CSS are collected during initialization and assembled afterward.
- **Binding order:** Registered binding functions and component objects are processed in registration order.
- **GF.Dialog timing:** The public dialog reference is assigned after the binding phase and before `InitializeDialog()`.
- **Document readiness:** A constructed dialog object does not guarantee that the embedded browser has loaded its HTML.
- **Runtime callbacks:** Ordinary LuaButton handlers execute while the dialog is open; they are separate from result-processing callbacks.
- **Dialog result:** `ShowDialog()` determines whether `ProcessResults()` or `DiscardResults()` executes.
- **Ignored return values:** The framework ignores return values from the result-processing and finalization callbacks.
- **Finalization:** `FinalizeScript()` runs after normal dialog completion, not after every possible early exit or error.
- **Component cleanup:** Gadget Factory invokes registered component `Done()` methods during normal cleanup.
- **Temporary file:** `GadgetTemp.html` is normally removed after component cleanup.
- **Error protection:** Gadget Factory's protected event dispatcher does not automatically protect the entire lifecycle.
- **No implicit application rollback:** Cancelling the dialog does not reverse modifications already performed by application callbacks while it was open.

Related topics

- GadgetDef (p. 10) - Configure gadget properties and lifecycle callbacks.
- InitializeScript (p. 34) - Prepare application state and construct the interface.
- InitializeDialog (p. 49) - Work with the native dialog after field registration.
- Default Buttons (p. 288) - Control native dialog completion.

- LoadComponents (p. 36) - Load reusable component libraries.
- AddBinding (p. 329) - Register native dialog fields.
- Event Handling (p. 320) - Handle interactive LuaButton events.
- Dynamic HTML (p. 364) - Modify HTML after the document becomes available.
- StatusBar (p. 298) - Provide runtime status feedback.
- FileList (p. 257) - Example of a component using lifecycle cleanup.

Summary

Gadget Factory manages the Vectric gadget lifecycle through a central entry point and a collection of programmer-defined callbacks.

The framework performs job validation, HTML document generation, native dialog creation, field binding, dialog display, result dispatching, and cleanup.

The programmer supplies the application-specific behavior at the appropriate stages.

The essential distinction is:

`InitializeScript()` constructs the interface. `InitializeDialog()` receives the bound dialog. Runtime callbacks handle interaction. `ProcessResults()` and `DiscardResults()` handle completion. `FinalizeScript()` performs normal finalization.

Gadget Factory coordinates the sequence so the programmer does not have to reproduce the machinery in every gadget.

REFERENCE 2.5

InitializeScript(job,script_path)

`InitializeScript()` is the primary initialization routine for a Gadget Factory gadget. It is responsible for preparing the gadget's operating environment and constructing its interface before the dialog is created.

The function is defined by the gadget programmer and registered through GF.GadgetDef (p. 10).

Syntax

```lua
local function InitializeScript(job, script_path)
    -- Initialization code
end
```

Parameters

Parameter	Description
`job`	The current Vectric job object when `GF.GadgetDef.Job` is enabled. Otherwise, the factory does not create or validate a job object.
`script_path`	The gadget's script directory, supplied by Vectric through `main()`.

Both parameters are optional in the function declaration if the gadget does not need them.

Responsibilities

Typical initialization tasks include:

- Establishing initial values and examining the current job.
- Loading component libraries through GF.LoadComponents() (p. 36).
- Defining CSS through GF.AddStyle() (p. 108).
- Constructing HTML content using GF.AddElements() (p. 74) and the individual field helpers.
- Creating component instances and registering event handlers.

Example: Sectio Divina

Sectio Divina uses `InitializeScript()` to examine the current selection, establish its initial dimensions, load the required components, and construct its interface.

The following abbreviated example illustrates the structure:

```lua
local function InitializeScript(job)
    Job = job

    -- Examine the current selection
    -- Establish initial dimensions

    GF.LoadComponents {"RadioGroup","ButtonGroup"}
```

LUA / CONTINUED

```lua
    GF.AddStyle("body {background-color: #f7f0d9;}")

    -- Construct the interface
    -- using GF.AddElements()
end
```

The function is registered in the gadget definition:

LUA

```lua
GF.GadgetDef = {
    Job     = true,
    Width   = 500,
    Height  = 350,
    Title   = "Sectio Divina",

    InitializeScript = InitializeScript,
    InitializeDialog = InitializeDialog,
    ProcessResults   = ProcessResults,
}
```

Execution

Gadget Factory automatically invokes `InitializeScript(job, script_path)` after validating the required job, if any, and before generating the HTML document.

When initialization completes, Gadget Factory creates the `HTML_Dialog`, registers its collected bindings, and invokes InitializeDialog (p. 49) if that callback is provided.

Aborting initialization

Returning `false` explicitly prevents the dialog from being created.

LUA

```lua
local function InitializeScript(job)
    if not SomeRequiredCondition then
        return false
    end

    -- Continue initialization
end
```

An explicit `true` return is not required. The factory proceeds unless the function returns exactly `false`.

Related topics

- GadgetDef (p. 10)
- InitializeDialog (p. 49)
- LoadComponents (p. 36)
- AddElements() (p. 74)
- ProcessResults (p. 25)

REFERENCE 2.6

LoadComponents()

`GF.LoadComponents()` loads one or more Gadget Factory component libraries and makes their public interfaces available through the `GF` table.

Gadget Factory uses a modular architecture. Rather than loading every available component when a gadget starts, the programmer specifies only those libraries required by the application.

Each library is loaded from Gadget Factory's `LIBs` directory, initialized with access to the framework's internal component services, and registered under its exported name.

Libraries may also register companion CSS stylesheets, which Gadget Factory incorporates into the generated HTML document.

Syntax

```lua
GF.LoadComponents(components)
```

Parameters

Parameter	Description
`components`	An array-style Lua table containing the names of the libraries to load.

Return value: The `GF` table, with the requested libraries' public exports available through it. This is the existing framework table, not a new table of exports.

Returning `GF` also supports component reuse, including LayerPicker's (p. 252) use of DataPicker (p. 246). Ordinary gadget code can ignore the return value and continue using its existing `GF` variable.

Basic usage

```lua
GF.LoadComponents {"RadioGroup", "CheckboxGroup"}
```

This loads both libraries and makes their constructors available:

```lua
GF.RadioGroup.New(...)
GF.CheckboxGroup.New(...)
```

The alternative parenthesized syntax is equally valid:

```lua
GF.LoadComponents({"RadioGroup", "CheckboxGroup"})
```

Lua permits parentheses to be omitted when a function receives a single table constructor as its argument.

The shorter form is used throughout Gadget Factory's examples.

Available libraries

The following libraries are included in the component collection documented in this reference.

Library	Public interface
RadioGroup	GF.RadioGroup
CheckboxGroup	GF.CheckboxGroup
ButtonGroup	GF.ButtonGroup
ToolStrip	GF.ToolStrip
Combobox	GF.Combobox
ListBox	GF.ListBox
DataPicker	GF.DataPicker
LayerPicker	GF.LayerPicker
FileList	GF.FileList
StatusBar	GF.StatusBar

Most provide reusable component instances created through `New()`.

StatusBar is an exception. It is a singleton utility initialized through `GF.StatusBar.Init()`.

All are loaded through the same mechanism, regardless of how their public APIs are organized.

See Components (p. 113) and StatusBar (p. 298) for their individual usage.

When to load components

Component libraries should normally be loaded during InitializeScript (p. 34), before constructing any instances that depend on them.

For example:

```lua
local OrientationGroup

local function InitializeScript(job)

    GF.LoadComponents {"RadioGroup", "ButtonGroup"}

    OrientationGroup = GF.RadioGroup.New("Orientation")
```

LUA / CONTINUED

```
        :WithContent({
            {"Portrait",  1},
            {"Landscape", 2}
        }, 30, 140, 300)
end
```

The library must be loaded before its exported constructor can be accessed.

Attempting to use `GF.RadioGroup.New()` before loading RadioGroup will fail because the corresponding entry does not yet exist in the `GF` table.

Loading multiple libraries

A single call can load any number of libraries:

LUA

```
GF.LoadComponents {
    "RadioGroup",
    "CheckboxGroup",
    "ButtonGroup",
    "StatusBar"
}
```

Libraries are processed in the order in which they appear.

This makes it convenient to establish the gadget's requirements in one location near the beginning of initialization.

Loading only what is needed

A gadget using only RadioGroup does not need to load CheckboxGroup, FileList, or the other libraries.

LUA

```
GF.LoadComponents {"RadioGroup"}
```

This keeps initialization selective and avoids incorporating unnecessary component stylesheets or executing unrelated module initialization code.

Source and compiled modules

Gadget Factory supports two module file extensions:

Extension	Purpose
`.gfm`	Gadget Factory source module
`.gfc`	Compiled Gadget Factory module

Both are loaded through Lua's `loadfile()` mechanism.

The extensions distinguish the two forms of a component without requiring different public APIs.

File resolution

For each requested component, Gadget Factory searches the `LIBs` directory.

For example:

```lua
GF.LoadComponents {"RadioGroup"}
```

constructs the base pathname:

```text
...\Vectric Files\Gadgets\LIBs\RadioGroup
```

It then applies this precedence:

1. If `RadioGroup.gfc` exists, load it.
2. Otherwise, attempt to load `RadioGroup.gfm`.

The relevant implementation is:

```lua
local basepath = libpath .. name
local filepath

if Core.FileExists(basepath .. ".gfc") then
    filepath = basepath .. ".gfc"
else
    filepath = basepath .. ".gfm"
end
```

This allows the same gadget code to work with either source or compiled versions of the component library.

Compiled modules take precedence

When both `.gfc` and `.gfm` exist, the compiled `.gfc` file is always selected.

Gadget Factory does not compare modification dates or determine which file is newer.

For example:

```text
LIBs\
    RadioGroup.gfm
    RadioGroup.gfc
    RadioGroup.css
```

The loader uses `RadioGroup.gfc`.

This is important during development.

If the source module has been modified but an older compiled module remains in the directory, Gadget Factory will continue loading the compiled version.

To test changes to a source module, remove its corresponding `.gfc` file or regenerate that compiled file.

This prevents a surprisingly common source of confusion: editing Lua source successfully while observing no change in the running gadget.

Missing or invalid modules

When a compiled module is unavailable, Gadget Factory attempts to load the source module.

If the selected file does not exist or cannot be loaded, Lua raises an error through the loader's assertion.

An invalid compiled module does not automatically cause a fallback to source.

The fallback occurs only when the `.gfc` file is absent.

The current loader is deliberately simple; it does not implement version checking, recovery, or automatic source compilation.

How libraries expose their APIs

A Gadget Factory library is an ordinary Lua module that returns a table containing the public objects or functions it wishes to expose.

For example, the RadioGroup module ends with:

```lua
return {
    RadioGroup = RadioGroup
}
```

The loader retrieves this table and copies its entries into Gadget Factory's public `GF` table.

The relevant implementation is:

```lua
local exported = assert(loadfile(filepath))(Core)

for key, value in pairs(exported) do
    GF[key] = value
end
```

After RadioGroup has loaded, the statement:

```lua
GF.RadioGroup.New(...)
```

becomes available.

Exporting multiple names

The mechanism is not restricted to one exported name per library.

A module could return:

```lua
return {
    MyComponent = MyComponent,
    MyUtility   = MyUtility
}
```

Both names would become available through `GF`.

This is a general property of the loader, although the standard component libraries normally expose their primary component or singleton under one name.

Public-name collisions

The loader copies exported values directly into the `GF` table.

It does not check whether an exported name already exists.

Consequently, a library that exports an existing public name could overwrite that entry.

Programmers developing additional libraries should choose public names carefully and avoid collisions with Gadget Factory's established API.

The private Core interface

When Gadget Factory loads a component, it supplies a private `Core` table as an argument.

The component receives it using Lua's variable-argument mechanism:

```lua
local Core = ...
```

This gives component libraries access to selected framework services without exposing those services as ordinary gadget APIs.

The current Core interface includes:

Function	Purpose
`ComponentId`	Generate an element identifier associated with a component instance.
`InstanceCounter`	Create an independent counter for component instances.
`RegisterContent`	Register generated HTML and the component's native binding object.
`AddCSS`	Register a stylesheet for inclusion in the dialog.
`FileExists`	Test whether a file can be opened.
`AddHandler`	Register a Lua callback through Gadget Factory's dispatcher.
`NumberFormatter`	Create a reusable numeric formatting function.
`EscapeHtml`	Escape text for generated HTML, including attribute values.

These services are primarily intended for component-library development.

They are not automatically exposed as `GF.Core` or as independent public methods.

Why distinguish Core from GF?

Gadget Factory presents two different interfaces:

`GF` is the public interface used by gadget programmers.

`Core` is the internal service interface supplied to component libraries.

This separation allows components to use framework mechanisms such as automatic identifiers, HTML registration, and event dispatching without making those implementation facilities part of every gadget's public API.

It also allows the framework to evolve its internal architecture independently of ordinary component usage.

Example: RadioGroup initialization

The RadioGroup module begins with:

```lua
local Core = ...

Core.AddCSS("RadioGroup")

local NextInstance = Core.InstanceCounter()
```

The first statement receives the private framework services.

The second registers the component's stylesheet.

The third creates a counter used to generate independent RadioGroup instance IDs.

These operations occur when the module is loaded, before the gadget constructs its first RadioGroup instance.

The programmer does not need to perform them manually.

Automatic stylesheet registration

Most Gadget Factory components provide a companion CSS file.

For example:

```text
LIBs\
    RadioGroup.gfm
    RadioGroup.css
```

The module registers its stylesheet through:

```lua
Core.AddCSS("RadioGroup")
```

Gadget Factory retains the stylesheet name in its internal CSS collection.

When the dialog document is generated, Gadget Factory creates the corresponding HTML stylesheet link.

Conceptually:

```HTML
<link rel="stylesheet" href="...\LIBs\RadioGroup.css">
```

The final pathname is constructed from Gadget Factory's library directory.

Stylesheet loading order

Gadget Factory begins with its shared `Gadgets.css` stylesheet.

Component stylesheets are then registered in the order their modules request them.

For example:

```LUA
GF.LoadComponents {
    "RadioGroup",
    "CheckboxGroup",
    "ButtonGroup"
}
```

normally results in stylesheet links ordered as follows:

```CSS
Gadgets.css
RadioGroup.css
CheckboxGroup.css
ButtonGroup.css
```

Gadget-specific styling supplied through AddStyle (p. 108) is placed in the document's inline `<style>` block after the external stylesheet links.

This allows the programmer to override component styling where the normal CSS cascade permits it.

An important distinction

`LoadComponents()` **does not automatically look for a CSS file merely because the component was loaded.**

The component itself must register the stylesheet using `Core.AddCSS()`.

This makes stylesheets optional from the loader's perspective.

A library that does not require an external stylesheet may omit that registration.

Likewise, a library can register an appropriately named shared stylesheet if its implementation requires one.

Duplicate library loading

Gadget Factory tracks loaded library names in a private table named `LoadedComponents`.

When a module loads successfully, its name is recorded.

If a subsequent call requests the same name, Gadget Factory displays a warning instead of executing the library again.

For example:

```lua
GF.LoadComponents {"RadioGroup"}

-- Later:
GF.LoadComponents {"RadioGroup"}
```

The second call displays:

```text
Component library already loaded: RadioGroup
```

Why reject duplicate loads?

A component module may establish shared counters, register stylesheets, or create internal state when it executes.

Loading it again could reset those facilities or introduce duplicate registrations.

Gadget Factory therefore treats loading as a one-time operation for each library name.

This is distinct from creating multiple instances of a component.

```lua
GF.LoadComponents {"RadioGroup"}

local GroupA = GF.RadioGroup.New("First Group")
local GroupB = GF.RadioGroup.New("Second Group")
```

This is valid and expected.

The library is loaded once, but its constructor may be called repeatedly.

Duplicate names in one call

The same rule applies when a name appears twice within the supplied table.

```lua
GF.LoadComponents {
    "RadioGroup",
    "CheckboxGroup",
    "RadioGroup"
}
```

The second `RadioGroup` entry produces the duplicate-load warning.

For normal gadget development, declare each required library only once.

Dependency management

The loader itself does not automatically resolve dependencies between component libraries. A component can explicitly load its own dependencies: LayerPicker (p. 252), for example, loads DataPicker automatically.

List the libraries the gadget uses directly, following each component's loading instructions. There is no need to add DataPicker solely to satisfy LayerPicker's internal dependency.

This is particularly relevant when integrating Lua snippets exported by Forge. Their dependency comments identify the required libraries, but the programmer is responsible for incorporating those names into the gadget's existing `LoadComponents()` call.

Avoid adding a separate load call for every pasted snippet when several snippets depend on the same module.

Example: Sectio Divina

Sectio Divina provides a compact example of loading only the required libraries.

During InitializeScript (p. 34), the gadget executes:

```lua
-- Fire up the factory
GF.LoadComponents {"RadioGroup","ButtonGroup"}
```

It subsequently creates an orientation selector:

```lua
OrientationGroup = GF.RadioGroup.New(
    "Orientation", OrientationChanged
)
    :WithContent({
        {"Portrait",  1},
        {"Landscape", 2}
    }, 30, 141, 308)
    :Select(Orientation)
```

And a custom action group:

```lua
GF.ButtonGroup.New(5, 1, "sdButtons")
    :WithContent({
        {"Revert",  btnRevert},
        {"Undo",    btnUndo},
        {"Apply",   btnApply},
        {"Commit",  "ButtonOK"},
        {"Discard", "ButtonCancel"}
    }, 370, 5)
```

The two required component libraries were loaded once, before either component was constructed.

Their public constructors and stylesheets became available automatically.

Sectio Divina did not need to load unrelated components such as FileList, Combobox, or StatusBar.

This illustrates the intended use of selective library loading.

How LoadComponents works internally

The complete implementation is short enough to reproduce here:

```lua
GF.LoadComponents = function(components)
    for _, name in ipairs(components) do
        if not LoadedComponents[name] then

            local basepath = libpath .. name
            local filepath

            if Core.FileExists(basepath .. ".gfc") then
                filepath = basepath .. ".gfc"
            else
                filepath = basepath .. ".gfm"
            end

            local exported = assert(loadfile(filepath))(Core)

            for key, value in pairs(exported) do
                GF[key] = value
            end

            LoadedComponents[name] = true
        else
            DisplayMessageBox(
                "Component library already loaded: " .. name
            )
        end
    end
    return GF
end
```

Despite its small size, the function performs several distinct operations:

1. Traverses the requested library names.
2. Checks whether each library has already loaded.
3. Resolves the preferred source or compiled file.
4. Executes the selected module with Gadget Factory's private `Core` services.
5. Copies the module's exported API into `GF`.
6. Records the library as loaded.
7. Returns the updated `GF` table after processing the requested libraries.

The implementation relies on ordinary Lua features: tables, functions, iteration, and dynamic file loading.

There is no separate component registry or external configuration file required for the loader.

Why not use Lua's require()?

Gadget Factory uses explicit file loading rather than relying on Lua's ordinary `require()` module-search mechanism.

This gives the framework direct control over the library directory and the choice between `.gfm` and `.gfc`.

It also allows Gadget Factory to pass its private `Core` table directly into each loaded module.

The library namespace is therefore managed by Gadget Factory itself rather than by Lua's conventional package-loading infrastructure.

Important considerations

- **Load before construction:** Component libraries must be loaded before their exported functions or constructors are used.
- **Use a sequence table:** The `components` argument must contain sequentially indexed library names. The loader traverses it using `ipairs()`.
- **Use module names:** Supply names such as `"RadioGroup"`, not filenames such as `"RadioGroup.gfm"`.
- **Library location:** Modules are resolved relative to Gadget Factory's `LIBs` directory.
- **Extension preference:** `.gfc` takes precedence over `.gfm` whenever the compiled file exists.
- **No modification-date comparison:** The loader does not determine whether a compiled file is older than its source.
- **No recovery fallback:** A present but invalid `.gfc` file does not automatically trigger loading of `.gfm`.
- **One load per name:** Duplicate requests display a warning rather than loading the module again.
- **Multiple instances:** Loading a library once does not limit the number of component instances that can be created.
- **Public exports:** Returned module-table entries are copied directly into `GF`.
- **Name collisions:** Exported names are not checked for conflicts with existing Gadget Factory members.
- **Private services:** The `Core` table is supplied to component libraries, not exposed as part of the ordinary gadget API.
- **CSS registration:** Companion stylesheets must be registered by their modules; the loader does not discover CSS files automatically.
- **Loading order:** Libraries execute in the order requested, and their registered stylesheets follow that order.
- **No automatic dependency resolution:** Gadgets must load the component libraries they require.
- **Error handling:** Module-loading errors are not handled by the protected LuaButton callback dispatcher. The current loader does not wrap them in `pcall()`.
- **No unloading API:** The framework does not provide a public method for unloading or reloading an already loaded library.
- **Initialization timing:** Load libraries during `InitializeScript()`, before the dialog's HTML and bindings are assembled.

Related functions

- Components (p. 113) - Introduction to the reusable component library.
- InitializeScript (p. 34) - Prepare the gadget and construct its interface.
- AddStyle (p. 108) - Supply gadget-specific CSS overrides.
- AddBinding (p. 329) - Understand native Vectric field registration.

- Event Handling (p. 320) - Understand Gadget Factory's callback dispatcher.
- RadioGroup (p. 120) - Create mutually exclusive selection groups.
- CheckboxGroup (p. 137) - Manage Boolean option groups.
- ButtonGroup (p. 156) - Create grouped action buttons.
- ToolStrip (p. 173) - Create docked command strips.
- Combobox (p. 192) - Create dynamically managed dropdowns.
- ListBox (p. 217) - Create dynamically managed list controls.
- FileList (p. 257) - Create filesystem-backed selection controls.
- StatusBar (p. 298) - Initialize the singleton status display.

Summary

`GF.LoadComponents()` provides selective access to Gadget Factory's reusable component libraries.

It resolves modules from the framework's library directory, prefers compiled `.gfc` files over source `.gfm` files, supplies internal framework services, incorporates each module's public exports into the `GF` interface, and returns that same `GF` table.

Component stylesheets are registered automatically by the modules that require them.

Loaded libraries are tracked to prevent repeated initialization.

Load the libraries once. Construct as many component instances as the gadget requires.

The result is a modular framework in which gadget programmers use a consistent public interface without needing to manage the underlying component-loading machinery.

REFERENCE 2.7

InitializeDialog(dlg)

Optional. Called after Gadget Factory creates the native dialog and applies its field bindings, but before displaying it.

Its primary purpose is to provide a convenient way to retain a local reference to the dialog, avoiding repeated references to `GF.Dialog`.

For example:

```lua
local Dialog

local function InitializeDialog(dlg)
    Dialog = dlg
end
```

The dialog object can subsequently be accessed through the local `Dialog` variable.

Important: Do not rely on DOM-dependent updates from InitializeDialog().

The native dialog and its field bindings exist at this point. However, in native Aspire verification, calling `GF.ExecutionEnabled(true)` from `InitializeDialog()` failed to enable the Execute button. That operation relies on `SetOuterHtml()`.

The useful finding is that DOM modification cannot be relied upon at this lifecycle stage. It does **not** conclusively establish that the DOM is absent, or explain exactly how the browser handles an early modification.

Arrange interface work at the appropriate stage:

- **Before dialog creation:** Use `InitializeScript()` and Gadget Factory's interface-building functions to define the initial contents.
- **After the HTML document has loaded:** Perform DOM-dependent updates from a callback. See Post-load initialization (p. 367).

Use `InitializeDialog()` to obtain access to the bound dialog. Construct the initial interface in `InitializeScript()`, and defer DOM-dependent updates until a callback after document loading.

REFERENCE 2.8

Factory Functions

This is an inventory of the functions and facilities defined in `GadgetFactory.gfm`, organized by purpose and intended audience.

Gadget Factory has four categories of functions:

1. **Public API** - Functions intended for ordinary gadget programming.
2. **Advanced API** - Exposed functions for programmers who need direct access to the framework's machinery.
3. **Component API** - Functions supplied to component libraries through `Core`.
4. **Internal functions** - Implementation details managed entirely by Gadget Factory.

A function being accessible does not necessarily mean it is intended for general use.

1. Public API - Gadget Programming

These are the functions a gadget programmer would ordinarily use.

Framework and Dialog Management

Function	Purpose
`GF.LoadComponents()`	Loads requested component libraries and makes their exported members available through `GF`.
`GF.ExecutionEnabled()`	Gets or changes the enabled state of the standard Execute button.
`GF.NoDefaultButtons()`	Suppresses the default Execute and Cancel buttons, allowing a gadget or component to supply its own.
`GF.AddStyle()`	Adds custom CSS to the generated HTML document.
`GF.NumberFormatter()`	Creates a number-formatting function with the requested number of decimal places, removing unnecessary trailing zeros.

HTML and Layout

Function	Purpose
`GF.AddElement()`	Adds a string of HTML to the dialog.
`GF.AddElements()`	Processes a table of HTML strings, nested tables, and function calls to construct an interface.
`GF.AddDiv()`	Creates a containing HTML `div`, optionally including attributes and nested elements.
`GF.AddControlLabel()`	Creates an HTML label, optionally associated with a field and positioned within the dialog.
`GF.EscapeHtml()`	Escapes text before it is incorporated into HTML. See the Preview Workbench (p. 400) for the canonical utility in use.

Vectric Fields and Controls

These functions generate the appropriate HTML and arrange the necessary Vectric field bindings.

Function	Purpose
`GF.AddLabelSpan()`	Adds a span that can be updated through Vectric's label-field interface.
`GF.AddLabelField()`	Adds a Vectric-compatible label field using a `div` element.
`GF.AddDoubleField()`	Adds a numeric field for floating-point values, with optional change notification.
`GF.AddTextField()`	Adds a text input field, with optional change notification.
`GF.AddIntegerField()`	Adds an integer input field, with optional change notification.
`GF.AddStateFlag()`	Adds a hidden checkbox bound to a Vectric Boolean field.

Important distinction: `GF.AddControlLabel()` creates ordinary HTML. `GF.AddLabelSpan()` and `GF.AddLabelField()` also register Vectric bindings.

Public Data

Not everything in `GF` is a function.

Member	Purpose
`GF.GadgetDef`	The gadget's definition, supplied by its main script. See GadgetDef (p. 10).
`GF.Dialog`	Reference to the created `HTML_Dialog` object. Available after dialog creation.

Member	Purpose
`GF.OnError`	The active error-reporting function, initially assigned a default reporter and optionally replaced through `GadgetDef.OnError`.

Loading components may also add new members to `GF`. These belong in the documentation for their respective components.

2. Advanced API - Exposed Machinery

These functions are available through `GF`, but are generally unnecessary for ordinary gadget programming.

Callback and Binding Management

Function	Purpose
`GF.AddHandler()`	Registers a callback under a control identifier. Rejects an identifier that already has a handler.
`GF.ReplaceHandler()`	Replaces or removes a callback associated with an identifier.
`GF.AddBinding()`	Registers a function to be called when Gadget Factory binds the dialog's controls.

These functions provide access to machinery normally managed automatically by Gadget Factory.

Most gadgets should use the callbacks supported by their fields and components instead.

Read-Only HTML Generation

These functions return HTML without adding it to the dialog or registering Vectric bindings.

Function	Purpose
`GF.GetElementsHtml()`	Produces HTML from nested element definitions using registered read-only renderers.
`GF.GetDivHtml()`	Returns the HTML for a containing `div`.
`GF.GetLabelHtml()`	Returns ordinary HTML label markup.
`GF.GetLabelSpanHtml()`	Returns label-span markup.
`GF.GetLabelFieldHtml()`	Returns label-field markup.
`GF.GetDoubleFieldHtml()`	Returns double-field markup.
`GF.GetTextFieldHtml()`	Returns text-field markup.
`GF.GetIntegerFieldHtml()`	Returns integer-field markup.

Function	Purpose
`GF.GetStateFlagHtml()`	Returns hidden state-flag markup.

These functions are particularly useful for advanced interface generation and inspection.

`GF.GetElementsHtml()` supports the Factory's registered element functions. It is not a general-purpose mechanism for executing arbitrary component functions.

3. Component API - Core

`Core` is a local table containing services intended for reusable component libraries.

It is passed to libraries by `GF.LoadComponents()`, allowing components to use selected framework facilities without exposing all internal machinery.

Function	Purpose
`Core.ComponentId()`	Creates an element identifier using a component instance's ID.
`Core.InstanceCounter()`	Creates a counter function that generates successive instance numbers.
`Core.RegisterContent()`	Registers a component's HTML and queues the component for dialog binding.
`Core.AddCSS()`	Adds a component stylesheet to the list included in the generated document.
`Core.FileExists()`	Tests whether a file exists.
`Core.AddHandler()`	Registers a callback using the same handler machinery exposed by `GF.AddHandler()`.
`Core.NumberFormatter()`	Uses the same number formatter exposed through `GF.NumberFormatter()`.
`Core.EscapeHtml()`	Provides the shared HTML escaping utility used by component libraries. Gadget code uses `GF.EscapeHtml()` (see ToolStrip's attribute escaping (p. 186)).

`Core` is not returned to ordinary gadget scripts. Its purpose is to supply component authors with the additional facilities their libraries require.

4. Internal Functions and Machinery

These functions and structures belong to Gadget Factory's implementation. They are not intended to be accessed directly by gadget programmers or components.

HTML Construction

Function or Structure	Purpose
`DOM()`	Assembles and writes the complete temporary HTML document.
`DivHtml()`	Constructs opening `div` markup.
`FieldHtml()`	Generates HTML for Vectric-compatible input fields.
`AddField()`	Adds field HTML, registers its binding, and arranges optional change handling.
`BindLabel()`	Queues the Vectric binding for a label.
`ElementRenderers`	Associates Factory element functions with their read-only HTML renderers.

Internal HTML Helpers

The following functions generate markup used by both the `Add...` operations and their `Get...Html` counterparts:

- `LabelHtml()`
- `LabelSpanHtml()`
- `LabelFieldHtml()`
- `StateFlagHtml()`

Their implementations are local, but they are deliberately exposed through corresponding `GF.Get...Html` aliases.

Internal Field Definitions

Structure	Purpose
`DoubleField`	Defines how double fields are created, read, updated, and displayed.
`TextField`	Defines text-field creation and retrieval.
`IntegerField`	Defines integer-field creation and retrieval.
`D`	The formatter used by double fields.

Private State and Configuration

Member	Purpose
`LoadedComponents`	Records which component libraries have already been loaded.
`Dlg`	Internal reference to the Vectric dialog object.
`DOMfile`	Path of the temporary HTML document.
`CSS`	Ordered collection of stylesheet names.
`StyleHTML`	Collection of custom inline CSS.
`BodyHTML`	Accumulated HTML for the dialog body.
`bindings`	Queued Vectric binding functions and component objects.
`dispatch`	Maps callback identifiers to their Lua handlers.
`DefaultButtons`	HTML for the standard Execute and Cancel buttons.
`HeadHTML`	HTML document header.
`DefaultOnError()`	Default error-reporting function used by the callback mechanism.
`Done()`	Private framework cleanup routine, called after the gadget's optional `FinalizeScript()` hook.

The following internal helper implementations have explicitly exposed aliases: `NumberFormatter`, `ReplaceHandler`, `AddHandler`, `AddBinding`, and `AddElement`. Their visibility is governed by those aliases rather than their local declarations.

5. Public Exposures and Aliases

Some functions are deliberately defined as local functions and subsequently made available through `GF`, `Core`, or both.

An alias is another name for the same function, not another implementation of it.

The source contains a section labeled `PUBLIC EXPOSURES` with these assignments:

```lua
GF.NumberFormatter = NumberFormatter
GF.ReplaceHandler = ReplaceHandler
GF.AddHandler = AddHandler
GF.AddBinding = AddBinding
GF.AddElement = AddElement
```

Two of these functions are also shared with component libraries:

```lua
Core.AddHandler = AddHandler
Core.NumberFormatter = NumberFormatter
```

In the actual `Core` table, these appear as table entries rather than subsequent assignments.

Four HTML helpers are exposed in the same fashion, although their assignments appear elsewhere in the source:

```lua
GF.GetLabelHtml = LabelHtml
GF.GetLabelSpanHtml = LabelSpanHtml
GF.GetLabelFieldHtml = LabelFieldHtml
GF.GetStateFlagHtml = StateFlagHtml
```

This approach allows the Factory to keep its internal definitions organized while deciding explicitly which functions are made available to other code.

Removing an alias can therefore withdraw a function from an exposed interface without necessarily removing its internal implementation.

6. Vectric Entry Points

Two global functions are defined by Gadget Factory because Vectric needs to call them directly.

main(script_path)

The entry point used by Vectric to launch a gadget.

It reads `GF.GadgetDef`, handles job requirements, calls the gadget's initialization hooks, creates and displays the dialog, processes the results, and performs cleanup.

Gadget programmers do not need to implement their own `main()`.

See Gadget Lifecycle (p. 15) and GadgetDef (p. 10).

OnLuaButton_XXXX(id)

The common entry point used by Vectric when a registered LuaButton is activated.

It locates the corresponding callback through the Factory's dispatch mechanism and invokes it with protected error handling.

Gadget programmers normally supply callback functions rather than defining their own `OnLuaButton_XXXX()`.

Both functions are global for Vectric's benefit, but are considered **framework-owned entry points**, not public functions intended for direct use.

Documentation Scope

The **Gadget Factory Programmer's Reference** brings the public API, components, and advanced topics together in a single publication.

The Components (p. 113) section explains the reusable components, including their use of the additional facilities available through `Core`.

The Advanced Topics (p. 316) section covers the advanced interfaces, implementation details, callback routing, HTML generation, and lifecycle management.

This organization allows programmers to understand and use Gadget Factory without first learning how all its internal machinery works.

CHAPTER 03

03

Conventions & contracts

The rules that make shared components predictable.

IN THIS CHAPTER

Programming Conventions 59

Framework Contracts 62

Vectric & Trident Considerations 66

REFERENCE 3.1

Programming Conventions

Gadget Factory encourages a consistent programming style without imposing unnecessary restrictions on how gadgets are written.

The conventions collected here reflect established practices used throughout the framework and its example gadgets. They are intended to improve readability, maintainability, and consistency rather than define mandatory syntax.

For technical requirements that must be observed, see Framework Contracts (p. 62).

Script organization

Convention	Detailed reference
Load Gadget Factory into the local variable `GF`.	Loading Gadget Factory (p. 9)
Define the gadget's configuration and callbacks through `GF.GadgetDef`.	GadgetDef (p. 10)
Use `InitializeScript()` for application setup and interface construction.	InitializeScript(job,script_path) (p. 34)
Load required component libraries together, preferably in one call.	Loading multiple libraries (p. 38)
Keep shared application variables and callbacks at script scope when necessary.	Lifecycle example (p. 29)
Reserve `InitializeDialog()` for operations requiring the native dialog object.	InitializeDialog(dlg) (p. 49)

Use descriptive variable names for important application objects. For example, `OrientationGroup` communicates more than `rg1`, particularly when several components are present.

Gadget Factory does not prescribe a naming convention for programmer-owned variables.

Components and methods

Component constructors generally use dot notation, while instance methods use colon notation.

```lua
local Options = GF.RadioGroup.New("Options")
    :WithContent(items, 20, 50, 250)
    :Select(1)
```

Singleton utilities, such as StatusBar, use their documented dot-notation APIs.

See Dot and colon conventions (p. 115).

Prefer the established `New()` and `WithContent()` pattern for ordinary component construction. Use `PrepareContent()` and `GetHtml()` independently only when their separation is actually needed.

See Preparing and rendering components (p. 117).

HTML construction

Gadget Factory's nested-table syntax is generally preferable to assembling large HTML strings manually when building interactive interfaces.

Use AddElements (p. 74) to combine raw HTML, nested containers, and element functions.

When omitting intermediate arguments from function-call tables, use `false` placeholders where appropriate rather than introducing `nil` gaps.

See Optional arguments (p. 76).

Favor the separation:

- **Lua:** Application behavior and state.
- **HTML:** Interface structure.
- **CSS:** Layout, presentation, and visual states.

See AddStyle (p. 108) and CSS-driven states (p. 341).

Values and callbacks

Use meaningful, stable keys and associated values rather than depending unnecessarily on displayed labels or control indices.

See Programmer-defined values (p. 116).

Programmatic selection changes generally do not invoke component selection callbacks. Invoke any required application logic explicitly.

See Component callbacks (p. 116).

Prefer concise callbacks that delegate substantial work to appropriately named functions.

Numeric formatting

For reusable numeric formatters, a short identifier that communicates precision is recommended:

```lua
local D3 = GF.NumberFormatter(3)
local D4 = GF.NumberFormatter(4)
```

These names are conventions, not framework requirements.

See Function factories (p. 282).

Choosing the simplest facility

Use the highest-level facility that adequately solves the problem.

A standard field primitive is generally preferable to a manually bound input. A component is generally preferable to duplicating its HTML and callback machinery.

However, Gadget Factory deliberately supports custom solutions when its predefined controls are insufficient.

The Advanced Topics (p. 316) section explains how to extend the framework without modifying its internals.

Consistency should make programming easier, never become an obstacle to solving the problem.

REFERENCE 3.2

Framework Contracts

Gadget Factory establishes several contracts governing how gadgets, components, native bindings, and callbacks interact.

Unlike the recommendations in Programming Conventions (p. 59), these contracts describe requirements or assumptions upon which the framework's operation depends.

Violating a contract may cause a Lua error, prevent a control from functioning, or produce unexpected behavior.

Framework ownership

Contract	Detailed reference
Gadget Factory owns the global `main()` entry point.	Vectric entry point (p. 17)
Gadget Factory owns the global `OnLuaButton_XXXX()` dispatcher.	Global dispatcher (p. 324)
Gadgets define their configuration through `GF.GadgetDef`.	GadgetDef (p. 10)
`InitializeScript()` is required by the current implementation.	InitializeScript (p. 18)

Ordinary gadget code should not replace the framework's global entry points.

Initialization and execution

The framework's lifecycle imposes several ordering requirements.

Contract	Detailed reference
Returning exactly `false` from `InitializeScript()` aborts initialization.	Aborting initialization (p. 20)
Native bindings execute after `HTML_Dialog` construction.	Binding phase (p. 22)
`GF.Dialog` is assigned only after native binding completes.	Dialog availability (p. 23)
DOM-dependent updates cannot be relied upon during `InitializeDialog()`; defer them to a callback after document loading.	DOM update timing (p. 366)

Contract	Detailed reference
Runtime field updates require an existing native binding.	Binding sequence (p. 331)

Native dialog creation and field binding do not establish readiness for DOM updates.

The native Aspire result establishes unreliable modification at this stage, not conclusive absence of the DOM.

Component registration

Contract	Detailed reference
Load a library before using its exported API.	Loading components (p. 37)
Each library name is loaded only once; duplicate requests produce a warning.	Duplicate loads (p. 43)
Compiled `.gfc` files take precedence over `.gfm` sources.	Module precedence (p. 38)
`WithContent()` normally installs a component once per instance.	Component construction (p. 115)
`GetHtml()` does not establish working native bindings or handlers.	HTML versus binding (p. 354)

Several newer components explicitly reject repeated `WithContent()` calls. Older components may not, but repeated registration can still cause duplicate controls or handlers.

Control identifiers

HTML identifiers connect the visible interface to Gadget Factory's binding and event systems.

Each relevant control must have a unique ID, and its native binding or handler must reference that ID correctly.

The framework does not perform comprehensive duplicate-ID validation for programmer-supplied HTML.

See Element identifiers (p. 369).

When changing an existing control's HTML, retaining its identifier may preserve its native binding. The supported experimental findings and their limitations are documented in Vectric & Trident Considerations (p. 66).

Event-handler contracts

`GF.AddHandler()` registers one handler per control ID. Registering an already occupied ID raises an assertion error.

`GF.ReplaceHandler()` can replace, establish, or remove a handler.

The dispatcher passes one optional argument to the registered function. It does not automatically supply the originating control ID or unpack argument tables.

See AddHandler (p. 320) and ReplaceHandler (p. 322).

Callback return values

A distinction must be observed between callbacks managed by Gadget Factory and callbacks defined directly for Vectric.

Callback mechanism	Return-value requirement
Function registered through `GF.AddHandler()`	No explicit `true` return required; Gadget Factory handles it.
Programmer-defined native Vectric LuaButton callback	Must satisfy Vectric's callback contract, including returning `true`.
`InitializeScript()`	Exactly `false` aborts initialization.
`ProcessResults()` and `DiscardResults()`	Return values are ignored by Gadget Factory.

See Callback return values (p. 327) and Processing results (p. 25).

Dialog completion and cleanup

The special `ButtonOK` and `ButtonCancel` identifiers participate in Vectric's native dialog-completion mechanism.

Gadget Factory's standard Execute button is initially inactive. Its availability is controlled through `GF.ExecutionEnabled()`.

See Default Buttons (p. 288).

Cancellation is not an automatic rollback.

Changes made during dialog execution are not automatically reversed when the user cancels.

Likewise, component cleanup may persist previously committed changes, as demonstrated by FileList's details editor (p. 267).

`FinalizeScript()` and the framework's cleanup routine execute during normal completion, but are not guaranteed after early exits or unhandled errors.

See Finalization (p. 27) and Component cleanup (p. 28).

Error handling

Gadget Factory executes registered LuaButton handlers through protected calls and directs intercepted errors to `GF.OnError`.

This protection does not automatically extend to lifecycle callbacks, native bindings, or arbitrary programmer-supplied Lua functions.

See Event error handling (p. 327) and Lifecycle error boundaries (p. 31).

Summary

Gadget Factory's contracts define the boundaries between programmer-owned application code and framework-managed infrastructure.

The most important are:

Respect the lifecycle, preserve control identities, register components and bindings correctly, and allow Gadget Factory to manage its own entry points and callback dispatcher.

Within those boundaries, programmers retain considerable freedom to design and extend their gadgets.

REFERENCE 3.3

Vectric & Trident Considerations

Gadget Factory operates within Vectric's `HTML_Dialog` environment, using an embedded browser based on Microsoft's Trident engine.

Although this environment supports substantial HTML, CSS, JavaScript, and Lua interaction, it has behaviors that differ from modern web browsers and conventional application frameworks.

During Gadget Factory's development, several important limitations and useful capabilities were established through experimentation in Aspire.

This chapter collects those findings and provides links to their detailed explanations.

Understanding the evidence

Two distinctions should be kept in mind.

Documented API behavior refers to functionality described by Vectric's native interface documentation or explicitly implemented by Gadget Factory.

Experimentally established behavior refers to results observed during our testing in Aspire and its embedded browser.

Experimental results are valuable working knowledge, but should not automatically be generalized to every Vectric version, control type, or Windows configuration.

LuaButton behavior

Vectric recognizes ordinary Lua callback controls through its special `LuaButton` class.

Our tests established that this class should be used without additional class names on the same element.

```HTML
<input type="button" class="LuaButton"
       id="btnApply" value="Apply">
```

For custom styling, target the element by ID or through a surrounding container rather than adding more classes.

See LuaButton requirements (p. 371).

Native Vectric callbacks require an appropriate return value. Gadget Factory's dispatcher handles that requirement automatically for registered handlers.

See Callback return values (p. 327).

Native `ButtonOK` and `ButtonCancel` completion is a separate mechanism. Raw HTML buttons with these identifiers worked without `LuaButton` in the Hello-Millie Aspire test (p. 289). That result does not remove the class requirement for ordinary Lua callback controls.

Document readiness

Creating the native dialog and applying its field bindings does not establish that DOM updates will work during `InitializeDialog()`.

Our earlier tests showed that `SetInnerHtml()` before `ShowDialog()` failed to populate the intended content, while the same operation worked from a callback after the document loaded.

Native Aspire verification also found that `GF.ExecutionEnabled(true)` from `InitializeDialog()` failed to enable Execute. That function uses `SetOuterHtml()`.

These findings establish that DOM modification is unreliable at this lifecycle stage. They do not conclusively prove that the DOM is absent.

See Dialog initialization (p. 366) and Post-load initialization (p. 367).

Practical rule: Use `InitializeScript()` to construct the original interface, then perform DOM-dependent updates from a callback after the HTML document has loaded.

Dynamic HTML and native bindings

One particularly useful experimental finding concerns preserved control identifiers.

Replacing an existing bound control with another element carrying the same ID allowed its previously established Vectric binding to remain functional in our tests.

See Replacing bound controls (p. 369).

This behavior is established for the tested same-ID scenarios.

It does **not** imply that newly introduced IDs automatically receive native bindings or that every field type behaves identically.

HTML state versus native state

Replacing a bound input with markup containing a new HTML `value` attribute did not reliably establish the corresponding native field value.

Use Vectric's native `Update...Field()` methods when changing registered field values.

See Field values and HTML replacement (p. 370).

Preserving event behavior

A replaced field may retain its native binding while losing its callback behavior if the replacement omits the required `onchange` attribute or hidden LuaButton bridge.

See Editable field event bridges (p. 372).

Preserving a binding and preserving its event mechanism are separate responsibilities.

CSS considerations

Vectric's Trident engine supports useful CSS facilities, including pseudo-classes and sibling selectors used throughout Gadget Factory.

However, newer browser features should not be assumed to work without verification.

Useful examples include:

CSS facility	Reference
`:checked` and sibling selectors	State-driven styling (p. 341)
`::-ms-check` for native radio appearance	RadioGroup styling (p. 133)
Custom dropdown arrow using `::-ms-expand`	Combobox styling (p. 212)
CSS-controlled button states	ToolStrip styling (p. 188)

The component stylesheets are designed for Vectric's embedded environment.

When introducing unfamiliar CSS properties, test their behavior in Aspire rather than relying solely on a modern browser preview.

Modal message boxes during callbacks

Experiments revealed that displaying a modal `MessageBox()` during interactive callbacks could interfere with pending dialog updates.

Observed effects included delayed or lost visible changes, focus irregularities, and queued user interactions.

See Dynamic HTML considerations (p. 376).

For routine feedback, use an in-dialog display such as StatusBar (p. 298) rather than repeatedly interrupting interaction with modal dialogs.

This does not prohibit message boxes. It means they should be used deliberately, particularly around operations that update controls.

Dynamically inserted JavaScript

Inserting HTML containing a `<script>` element does not guarantee that the script will execute.

For dynamic interfaces, define required JavaScript during initial document construction and let it operate on elements inserted later.

See Dynamic JavaScript (p. 375).

When inserting programmer-generated source code or other arbitrary text into HTML, treat it as data and escape it for the destination context.

Literal quotes, backslashes, newlines, and closing HTML tags can otherwise interfere with parsing.

Clipboard observations

Our native Aspire tests established that Trident's JavaScript clipboard facility could successfully copy Unicode text to the Windows clipboard.

The tested path preserved ordinary Unicode characters, variation selectors, supplementary Unicode characters, multiline text, quotes, backslashes, and literal closing HTML tags when pasted into VS Code.

The relevant JavaScript operation was:

```javascript
window.clipboardData.setData("Text", text);
```

This succeeded in the tested Aspire installation.

However, browser security settings may permit, prompt for, or deny clipboard access in other environments.

A separate experiment using Vectric's `VectricJob:CopyToClipboard()` produced incorrectly decoded Unicode text in the clipboard output.

These findings concern the tested environments and should not be treated as a universal guarantee of clipboard behavior.

Practical checklist

When developing unfamiliar or dynamic interface behavior:

1. **Confirm timing.** Is the native dialog constructed, and has the HTML document loaded?
2. **Confirm identity.** Are the relevant control IDs unique and preserved where necessary?
3. **Confirm binding.** Does Vectric have the required native field registration?
4. **Confirm event delivery.** Does the HTML retain the LuaButton class or event bridge required by the callback?
5. **Confirm presentation.** Does the CSS behave correctly in Trident?
6. **Test in Aspire.** Browser simulations are useful, but native verification remains the final authority for the intended host.

Summary

Vectric's embedded HTML environment is more capable than its conventional gadget examples might suggest.

Gadget Factory takes advantage of those capabilities while working around limitations discovered during development.

The most valuable lessons concern **document readiness, stable control identifiers, native field bindings, callback behavior, and browser-specific presentation**.

The detailed experiments and techniques are documented in Dynamic HTML (p. 364) and Event Handling (p. 320).

These findings provide practical guidance rather than a guarantee that every untested combination will behave identically.

We did the tinkering, so you don't have to.

Of course, adventurous programmers are always welcome to discover something new.

CHAPTER 04

04

HTML primitives

Build the interface from small, composable elements.

IN THIS CHAPTER

HTML primitives: overview 71

AddElement() 72

AddElements() 74

AddDiv() 80

AddControlLabel() 86

Field Primitives 93

AddLabelSpan() 103

AddStyle() 108

REFERENCE 4.1

HTML primitives: overview

Gadget Factory treats HTML construction as part of the Lua programming process rather than a separate document-authoring task. By combining structured tables, raw HTML, and executable element functions, a programmer can construct complete interactive interfaces directly in the gadget's source code, with minimal boilerplate and without manually managing HTML container boundaries.

AddElement() (p. 72) AddElements() (p. 74) AddDiv() (p. 80) AddStyle() (p. 108) AddLabelSpan() (p. 103) Field Primitives (p. 93)

- AddTextField (p. 98)
- AddDoubleField (p. 97)
- AddIntegerField (p. 98)
- AddLabelField (p. 99) AddControlLabel() (p. 86)

REFERENCE 4.2

AddElement()

`GF.AddElement()` inserts raw HTML markup into the gadget's dialog content.

It is the simplest HTML construction function exposed by Gadget Factory and provides the foundation for the higher-level HTML helpers.

Syntax

```lua
GF.AddElement(html)
```

Parameter

- `html` - A string containing HTML markup to insert into the dialog.

Return value: None.

Example

```lua
GF.AddElement('<h3>Surfacing Options</h3>')
```

Multiple HTML elements can be supplied in a single call:

```lua
GF.AddElement([[
<div class="instructions">
    <h3>Surfacing Options</h3>
    <p>Select a template before executing.</p>
</div>
]])
```

Behavior

`AddElement()` appends the supplied markup to Gadget Factory's internal HTML collection.

The markup is not rendered immediately. When the dialog is constructed, Gadget Factory combines the collected markup into the HTML document used by Vectric's `HTML_Dialog`.

Elements are included in the order in which they are added.

Important considerations

- The supplied HTML is inserted without modification or validation.
- HTML tags and attributes must be supplied by the programmer.

- The function does not automatically register Vectric fields, bindings, or event handlers.
- Content should be added before the dialog's HTML document is generated. Calling `AddElement()` after dialog creation does not update the displayed interface.

Relationship to AddElements

`GF.AddElement()` accepts a single HTML string, which may contain multiple HTML tags.

`GF.AddElements()` extends this capability by accepting a table containing HTML strings, nested element collections, and function calls.

The structured calling conventions of `AddElements()` are covered in the next chapter.

REFERENCE 4.3

AddElements()

`GF.AddElements()` constructs dialog content from a structured Lua table containing HTML strings, function calls, and nested element collections.

It extends AddElement() (p. 72) by allowing an entire section of an interface to be expressed in a single call.

Syntax

```lua
GF.AddElements(elements)
```

Parameter

- `elements` - An array-style Lua table containing HTML strings, function-call tables, nested element tables, or a mixture of these.

Return value: None.

Element types

`AddElements()` recognizes three kinds of entries.

Entry type	Behavior
String	Appends the string directly to the dialog HTML
Function-call table	Calls the function with the remaining table entries as arguments
Nested table	Processes the table recursively

Entries are processed sequentially, preserving their order.

1. HTML strings

A string is added directly to the dialog content.

```lua
GF.AddElements {
    "<h3>Dimensions</h3>",
    "<p>Enter the desired dimensions.</p>"
}
```

This is equivalent to two successive calls to `GF.AddElement()`.

Strings may contain complete HTML fragments, opening or closing tags, or ordinary text.

2. Function-call tables

A table whose first entry is a function is interpreted as a function call.

```lua
GF.AddElements {
    {GF.AddControlLabel, "Width:", "txtWidth"},
    {GF.AddDoubleField, "txtWidth", 12.0}
}
```

Each function-call table follows this convention:

```lua
{function, argument1, argument2, ...}
```

The function is supplied as a reference, not invoked when the table is constructed.

For example:

```lua
{GF.AddDoubleField, "txtWidth", 12.0}
```

is executed as:

```lua
GF.AddDoubleField("txtWidth", 12.0)
```

Do not add parentheses after the function name inside the element table.

3. Nested tables

A table whose first entry is not a function is treated as another element collection.

```lua
GF.AddElements {
    "<h3>Dimensions</h3>",
    {
        {GF.AddControlLabel, "Width:", "txtWidth"},
        {GF.AddDoubleField, "txtWidth", 12.0}
    },
    {
        {GF.AddControlLabel, "Height:", "txtHeight"},
        {GF.AddDoubleField, "txtHeight", 8.0}
    }
}
```

Nested tables are processed recursively and may contain additional nested tables.

Important: Nesting a Lua table does not automatically create an HTML container. It only groups entries for processing. Use `GF.AddDiv()` or explicit HTML tags when a corresponding HTML container is needed.

Working with optional arguments

One of the more useful conventions demonstrated by Sectio Divina involves passing optional arguments to field helpers.

Consider the signature:

```lua
GF.AddDoubleField(id, value, left, top, width, onchange)
```

Suppose a callback is required, but positioning and width arguments are not.

```lua
GF.AddElements {
    {GF.AddDoubleField, "txtX", X, false, false, false, XChanged}
}
```

The three `false` entries preserve the positional arguments without specifying coordinates or width.

This is important because Lua tables cannot reliably preserve array positions containing `nil` gaps.

The corresponding function call is:

```lua
GF.AddDoubleField("txtX", X, false, false, false, XChanged)
```

The callback is passed as a function reference. It will be invoked through Gadget Factory's field-handling mechanism when the field changes, not when `AddElements()` constructs the dialog.

Example: Sectio Divina dimensions

The following example is adapted from Sectio Divina's dimensions interface, illustrating the combination of HTML containers, labels, fields, and callbacks.

```lua
GF.AddElements {
    {GF.AddDiv, 'class="dimensions"', {
        {GF.AddControlLabel, "X:", "txtX"},
        {GF.AddDoubleField, "txtX", X,
            false, false, false, XChanged},

        {GF.AddControlLabel, "Y:", "txtY"},
        {GF.AddDoubleField, "txtY", Y,
            false, false, false, YChanged}
    }}
}
```

Here, `GF.AddDiv()` receives two arguments:

1. An HTML attribute string.

2. A nested table defining its content.

The outer `AddElements()` calls `AddDiv()`, which creates the opening `<div>`, processes its nested elements, and creates the closing `</div>`.

The resulting interface contains two labeled numeric fields enclosed in a single HTML container.

This arrangement keeps related interface elements together while leaving their visual presentation to CSS.

Mixing HTML and generated controls

Strings and function-call tables may appear together without additional grouping.

Another Sectio Divina pattern uses bound label spans separated by ordinary HTML text:

```lua
GF.AddElements {
    {GF.AddLabelSpan, "labX", D(X)},
    " × ",
    {GF.AddLabelSpan, "labY", D(Y)}
}
```

Here, `D()` represents the gadget's numeric formatting function (p. 281).

The two label spans are registered with Vectric, while the multiplication symbol is inserted as literal text.

This demonstrates an important capability: an element collection can combine dynamically bound controls with ordinary HTML without requiring every piece of content to have a dedicated helper.

Building element collections programmatically

Element tables are ordinary Lua tables. They can be assembled, extended, and reused before being passed to `AddElements()`.

```lua
local dimensions = {
    {GF.AddControlLabel, "X:", "txtX"},
    {GF.AddDoubleField, "txtX", X}
}

if IncludeY then
    dimensions[#dimensions + 1] =
        {GF.AddControlLabel, "Y:", "txtY"}

    dimensions[#dimensions + 1] =
        {GF.AddDoubleField, "txtY", Y}
end

GF.AddElements {
    "<h3>Dimensions</h3>",
    dimensions
}
```

This allows conditional sections and reusable element collections without constructing long HTML strings manually.

Execution behavior

`AddElements()` processes its table immediately when called.

For each entry, it performs one of the following operations:

- Appends an HTML string to the dialog's content collection.
- Executes a function with the supplied arguments.
- Recursively processes a nested element collection.

When a helper such as `GF.AddDoubleField()` executes, it creates its HTML and registers the appropriate Vectric binding. The actual binding occurs later, during dialog initialization.

The complete HTML document is generated after the gadget's initialization routine (p. 34) has populated the content collection.

Restrictions and considerations

- **Array-style tables:** Elements are traversed using `ipairs()`. Sequential integer indices are expected; a missing index can prevent subsequent elements from being processed.
- **Positional arguments:** Function-call tables use `table.unpack()` to pass arguments. Use `false` placeholders where appropriate instead of creating `nil` gaps.
- **Function references:** A function must occupy the first position of a function-call table to be executed. A bare function entry is not recognized.
- **Unsupported entries:** Values that are neither strings nor tables are silently ignored by `AddElements()`. Consequently, numeric values must be enclosed by a Lua string delimiter.
- **Raw HTML:** Strings are inserted without automatic escaping or validation.
- **Execution order:** Functions run in the order encountered, so side effects are possible. Functions are not restricted to Gadget Factory helpers, although those helpers are the intended use.
- **Timing:** Element collections should be processed during interface initialization, before the HTML document is generated.

Relationship to GetElementsHtml

`GF.GetElementsHtml()` accepts the same general nested-table structure but serves a different purpose.

Rather than executing the element helpers and registering bindings, it generates and returns an HTML string using their read-only rendering companions.

Only functions with registered renderers are supported.

This distinction matters when HTML markup is needed independently of the normal dialog construction process. In particular, generating field HTML with `GetElementsHtml()` does not register the corresponding Vectric field binding.

Related functions

- AddElement() (p. 72) - Insert a single HTML fragment.
- AddDiv() (p. 80) - Create an HTML container around an element collection.
- AddControlLabel() (p. 86) - Create an HTML label associated with a control.
- AddLabelSpan() (p. 103) - Create a bound inline label.
- AddDoubleField (p. 97) - Create a bound numeric field.
- GetElementsHtml() (p. 351) - Generate HTML from an element collection without executing its binding operations.

REFERENCE 4.4

AddDiv()

`GF.AddDiv()` creates an HTML `<div>` container and optionally populates it with an element collection.

It simplifies the construction of grouped interface elements by automatically generating the opening and closing `<div>` tags.

The function accepts standard HTML attributes and uses AddElements() (p. 74) to process its contents, including nested containers, raw HTML, and Gadget Factory element functions.

Syntax

```lua
GF.AddDiv(attributes, elements)
GF.AddDiv(elements)
```

Parameters

Parameter	Description
`attributes`	Optional string containing HTML attributes for the container.
`elements`	Optional Lua table containing the container's HTML elements and function calls.

Return value: None.

`AddDiv()` appends the generated HTML to Gadget Factory's dialog content collection. It does not return an HTML string.

Basic usage

A simple container can be created with:

```lua
GF.AddDiv('class="myPanel"', {
    "<h3>Panel Title</h3>",
    "<p>This is the panel content.</p>"
})
```

The resulting HTML is:

```html
<div class="myPanel">
    <h3>Panel Title</h3>
    <p>This is the panel content.</p>
</div>
```

The opening and closing `<div>` tags are supplied automatically.

Omitting attributes

When no attributes are required, the element collection can be passed as the first argument:

```lua
GF.AddDiv {
    "<h3>Dimensions</h3>",
    "<p>Enter the desired dimensions.</p>"
}
```

This produces an ordinary `<div>` without attributes.

Internally, `AddDiv()` recognizes that the first argument is a table and treats it as the element collection.

There is no need to write:

```lua
GF.AddDiv(nil, {
    "<h3>Dimensions</h3>"
})
```

Although this form is also valid.

Empty containers

Both parameters may be omitted:

```lua
GF.AddDiv()
```

This generates:

```html
<div></div>
```

Alternatively, attributes may be provided without content:

```lua
GF.AddDiv('id="placeholder"')
```

The resulting container is empty but retains the supplied attributes. This is useful for adding anchors whose content will be defined by a subsequent call to SetInnerHtml(id, html) (p. 364) after the dialog has been created.

Specifying HTML attributes

The `attributes` parameter is inserted directly into the opening `<div>` tag.

Any valid HTML attributes may be supplied, including `id`, `class`, `style`, and custom attributes.

For example:

```lua
GF.AddDiv('id="Dimensions" class="panel"')
```

Or with inline positioning:

```lua
GF.AddDiv(
    'style="position:absolute;left:50px;top:25px;"',
    {"<h3>Dimensions</h3>"}
)
```

Important: The parameter expects HTML attributes, not a bare CSS declaration.

For example:

```lua
-- Incorrect
GF.AddDiv("position:absolute;left:50px;", elements)

-- Correct
GF.AddDiv('style="position:absolute;left:50px;"', elements)
```

The programmer is responsible for supplying the appropriate attribute names, quotation marks, and CSS units.

Gadget Factory does not modify or validate the attribute string.

Using AddDiv with AddElements

`AddDiv()` is especially useful inside an AddElements() (p. 74) collection.

```lua
GF.AddElements {
    {GF.AddDiv, 'class="panel"', {
        "<h3>Dimensions</h3>",
        "<p>Enter the desired dimensions.</p>"
    }}
}
```

The first entry in the inner table is the function reference `GF.AddDiv`.

`AddElements()` recognizes it as a function-call table and passes the remaining entries as arguments:

```lua
GF.AddDiv('class="panel"', elements)
```

The function then creates the container and processes the supplied elements.

This is the same calling convention used by other Gadget Factory HTML helpers.

Nested containers

Because `AddDiv()` processes its contents through `AddElements()`, containers may be nested to any practical depth.

```lua
GF.AddElements {
    {GF.AddDiv, 'class="outerPanel"', {
        "<h3>Options</h3>",

        {GF.AddDiv, 'class="innerPanel"', {
            "<p>First group</p>"
        }},

        {GF.AddDiv, 'class="innerPanel"', {
            "<p>Second group</p>"
        }}
    }}
}
```

The outer container encloses two inner containers.

Each call to `AddDiv()` automatically supplies the appropriate opening and closing tags, eliminating the need to manage those boundaries manually.

Lua table nesting and HTML container nesting are distinct concepts. A nested table in `AddElements()` merely groups instructions; calling `AddDiv()` explicitly creates an HTML container.

Example: Sectio Divina

Sectio Divina uses nested `AddDiv()` calls to organize its dimensional history display.

The following excerpt is taken from its initialization routine:

```lua
GF.AddElements {
    {GF.AddDiv,
        'style="position:absolute;left:50px;top:25px;width:350px;"', {
        {GF.AddDiv, {
            [[<b style="margin-right:6px;">Original Dimensions:</b>]],
            {GF.AddLabelSpan, "labX0", D(X)}, " × ",
            {GF.AddLabelSpan, "labY0", D(Y)},
        }},
    }}
}
```

Several features are worth noting.

The outer container establishes the absolute position and width of the dimensional history panel.

The inner container groups the description and two dimension values. Because it requires no additional attributes, its element table is passed directly as the first argument.

The contents combine three different kinds of elements:

- Raw HTML defining the bold description.
- Two calls to `GF.AddLabelSpan()` that create bound label elements.
- A literal multiplication symbol between the dimensions.

The `D()` calls use the gadget's numeric formatting function (p. 281) to produce display strings.

The complete dimensional history section contains three such inner containers, representing the original, previous, and current dimensions.

Each is constructed using the same nesting convention.

Execution behavior

When called, `AddDiv()` performs three operations:

1. Appends an opening `<div>` tag containing the supplied attributes.
2. Processes the element collection, if supplied, using `GF.AddElements()`.
3. Appends the closing `</div>` tag followed by a newline.

These operations occur immediately during interface construction.

Although the generated HTML is collected for later use, functions inside the element collection execute as they are encountered.

For example, a call to `GF.AddLabelSpan()` within the container creates its HTML and registers the corresponding Vectric binding.

`AddDiv()` itself creates no Vectric binding.

Important considerations

- **Element collections must be tables.** To insert a single HTML string, enclose it in a table: `{"<p>Hello</p>"}`.
- **Attributes are optional.** Supplying an element table as the first argument automatically omits the attributes.
- **No automatic styling.** The container receives only the attributes explicitly supplied. Styling may also be applied through external or inline CSS.
- **Shared positioning convention.** The shared `Gadgets.css` stylesheet applies absolute positioning to div elements whose IDs begin with `div`. Use this prefix for containers intended to participate in that positioning convention. For ordinary flow-layout containers, choose another identifier or explicitly override the positioning.
- **No automatic escaping.** HTML strings and attributes are inserted without validation or sanitization.
- **Execution order matters.** Nested element functions execute while their enclosing container is being constructed.
- **Dialog construction:** Like other HTML primitives, `AddDiv()` is normally called during InitializeScript() (p. 34), before Gadget Factory generates the dialog document.

Relationship to GetDivHtml

Gadget Factory also exposes:

```lua
GF.GetDivHtml(attributes, elements)
```

This function accepts the same argument conventions but **returns an HTML string** instead of appending content to the dialog.

It uses `GF.GetElementsHtml()` to render the supplied element collection without executing the ordinary binding and registration operations.

Use `AddDiv()` for normal dialog construction. `GetDivHtml()` is appropriate when the generated markup itself is required.

Related functions

- AddElement() (p. 72) - Append raw HTML.
- AddElements() (p. 74) - Process structured HTML element collections.
- AddLabelSpan() (p. 103) - Create a bound inline label.
- GetElementsHtml() (p. 351) - Generate HTML without executing binding operations.
- InitializeScript (p. 34) - Construct the gadget interface before dialog creation.

REFERENCE 4.5

AddControlLabel()

`GF.AddControlLabel()` creates an HTML `<label>` element, optionally associating it with another HTML control.

It is primarily intended to provide descriptive text for editable fields such as AddDoubleField, AddTextField, and AddIntegerField (p. 93).

Unlike AddLabelField (p. 99) and AddLabelSpan() (p. 103), `AddControlLabel()` does not create a Vectric-bound field. It generates ordinary HTML, leaving the browser responsible for displaying the label and maintaining its association with the target control.

Syntax

```lua
GF.AddControlLabel(text, target, left, top)
```

Parameters

Parameter	Description
`text`	The text or HTML content displayed by the label.
`target`	Optional ID of the HTML control associated with the label.
`left`	Optional horizontal position in pixels.
`top`	Optional vertical position in pixels.

Return value: None.

The function appends the generated HTML to Gadget Factory's content collection.

Basic usage

```lua
GF.AddControlLabel("Width:", "txtWidth")
GF.AddDoubleField("txtWidth", 12.5)
```

The first call generates:

```html
<label for="txtWidth" style="white-space:nowrap;">Width:</label>
```

The corresponding field helper creates an HTML input with the ID `txtWidth`.

The `for` attribute establishes the association between the label and its target.

Associating labels with controls

A label's `target` must match the HTML `id` of the intended control.

```lua
GF.AddControlLabel("Height:", "txtHeight")
GF.AddDoubleField("txtHeight", 8.0)
```

Here, the label describes the field identified as `txtHeight`.

This association is part of standard HTML. In ordinary browser behavior, activating a label typically focuses its associated text input or toggles its associated checkbox.

`AddControlLabel()` does not implement additional Lua click handling. Any specialized label interaction must be implemented separately, and browser-specific behavior should be verified in Aspire.

Omitting the target

The target parameter may be omitted:

```lua
GF.AddControlLabel("Dimensions")
```

This generates:

```html
<label style="white-space:nowrap;">Dimensions</label>
```

The label is displayed but has no associated control.

Although this is supported, ordinary HTML elements such as `<span>`, `<div>`, or a heading are generally more appropriate for text that does not describe an input.

Positioning

Like the editable field primitives, `AddControlLabel()` supports optional absolute positioning.

```lua
GF.AddControlLabel("Width:", "txtWidth", 20, 50)
GF.AddDoubleField("txtWidth", 12.5, 100, 50, 120)
```

The label generates:

```html
<label for="txtWidth"
       style="position:absolute;left:20px;top:50px;white-space:nowrap;">Width:</label>
```

Both `left` and `top` must be supplied to enable absolute positioning.

Supplying only one coordinate does not establish absolute positioning.

When no coordinates are supplied, the label participates in the normal HTML layout.

Omitting positioning arguments

If the label is intended to follow the normal document flow, the positioning arguments may simply be omitted.

```lua
GF.AddControlLabel("Width:", "txtWidth")
```

When positional arguments must be preserved inside an AddElements() (p. 74) function-call table, `false` placeholders can be used:

```lua
GF.AddElements {
    {GF.AddControlLabel, "Width:", "txtWidth", false, false}
}
```

This is compatible with the nested-table conventions used throughout Gadget Factory.

Layout and styling

`AddControlLabel()` generates an inline HTML `<label>` element.

It includes one default CSS declaration:

```css
white-space: nowrap;
```

This prevents the label text from wrapping onto multiple lines.

The function does not automatically establish margins, alignment, or spacing between the label and its associated control.

Those responsibilities belong to the surrounding HTML and CSS.

Using CSS for layout

For interfaces containing several related fields, CSS is generally preferable to individually positioning each label and input.

```lua
GF.AddStyle([[
.dimensionRow {
    display: flex;
    align-items: center;
    gap: 8px;
```

LUA / CONTINUED

```
    margin-bottom: 6px;
}
]])
```

The corresponding interface can be constructed using:

LUA

```
GF.AddElements {
    {GF.AddDiv, 'class="dimensionRow"', {
        {GF.AddControlLabel, "Width:", "txtWidth"},
        {GF.AddDoubleField, "txtWidth", 12.5}
    }},
    {GF.AddDiv, 'class="dimensionRow"', {
        {GF.AddControlLabel, "Height:", "txtHeight"},
        {GF.AddDoubleField, "txtHeight", 8.0}
    }}
}
```

The CSS controls the arrangement of both rows, while each label retains its association with the appropriate field.

This separates interface structure from presentation and avoids maintaining individual coordinates.

CSS layout features must be supported by Vectric's embedded browser engine.

Example: Sectio Divina

Sectio Divina uses `AddControlLabel()` to identify its X and Y dimension fields.

The following excerpt comes from its initialization routine (p. 34):

LUA

```
GF.AddElements {
    {GF.AddDiv, 'class="cssTXTRow"', {
        {GF.AddControlLabel, "X:", "txtX"},
        {GF.AddDoubleField, "txtX", X,
            false, false, false, XChanged},
    }},
    {GF.AddDiv, 'class="cssTXTRow"', {
        {GF.AddControlLabel, "Y:", "txtY"},
        {GF.AddDoubleField, "txtY", Y,
            false, false, false, YChanged},
    }},
}
```

Several conventions are illustrated here.

Shared layout: Each label and field are enclosed in a container using the `cssTXTRow` class. The layout is controlled through CSS rather than individual field coordinates.

Control association: Each label's target matches the ID of its corresponding DoubleField.

Function-call tables: Both `AddControlLabel()` and `AddDoubleField()` are invoked through the nested-table syntax of `AddElements()`.

Callbacks: The DoubleFields have change callbacks, but the labels themselves do not participate in the event-dispatching mechanism.

This is the intended relationship between `AddControlLabel()` and Gadget Factory's editable fields: the label describes the control, while the field helper handles data binding and change notification.

AddControlLabel versus bound labels

Gadget Factory provides three functions whose names or behavior involve labels.

Despite their similarities, they serve different purposes.

Function	HTML element	Vectric binding	Purpose
`AddControlLabel`	`<label>`	No	Describe or identify an HTML control
`AddLabelField`	`<div>`	`AddLabelField()`	Display programmatically updateable text
`AddLabelSpan`	`<span>`	`AddLabelField()`	Display programmatically updateable inline text

The distinction is particularly important when updating a dialog.

For example:

```lua
Dlg:UpdateLabelField("labStatus", "Complete")
```

works with a label registered through `GF.AddLabelField()` or `GF.AddLabelSpan()`.

An `AddControlLabel()` label has no corresponding native Vectric label-field binding and cannot be updated through `UpdateLabelField()`.

Its content is ordinary HTML.

If the descriptive text must change dynamically, a bound label or an explicitly managed HTML element may be more appropriate.

GetLabelHtml

Gadget Factory also exposes a read-only companion:

```lua
GF.GetLabelHtml(text, target, left, top)
```

It accepts the same parameters but returns the generated HTML string instead of appending it to the dialog.

For example:

```lua
local html = GF.GetLabelHtml("Width:", "txtWidth")
```

The returned value is:

```html
<label for="txtWidth" style="white-space:nowrap;">Width:</label>
```

`GetLabelHtml()` does not modify the dialog content collection or create any bindings.

This renderer is also used by GetElementsHtml() (p. 351) when processing element collections containing `GF.AddControlLabel`.

Note that the public constructor is named `AddControlLabel`, while its rendering companion retains the name `GetLabelHtml`.

Execution behavior

When `GF.AddControlLabel()` is called, it:

1. Constructs an HTML `<label>` element.
2. Adds the `for` attribute when a target is supplied.
3. Adds optional absolute positioning when both coordinates are supplied.
4. Applies the `white-space:nowrap` declaration.
5. Appends the completed HTML to Gadget Factory's content collection.

These operations occur during interface construction.

Unlike the editable field primitives, `AddControlLabel()` does not register a deferred Vectric binding or a callback handler.

The generated label becomes part of the HTML document when Gadget Factory constructs the dialog.

Important considerations

- **Target IDs:** The `target` parameter should match the actual HTML `id` of the associated control.
- **Optional association:** A label can be created without a target, although ordinary HTML text elements are often preferable in that situation.
- **Positioning:** Both `left` and `top` are required for absolute positioning.
- **Default styling:** Labels include `white-space:nowrap` to prevent text wrapping.
- **No automatic spacing:** Margins, alignment, and spacing must be supplied through CSS or the surrounding HTML.

- **No Vectric binding:** The function does not register a Vectric LabelField.
- **No automatic event handler:** The function does not register Lua click callbacks or change handlers.
- **No automatic escaping:** Label text and target identifiers are inserted directly into HTML. Values originating from external input must be escaped appropriately.
- **Limited attributes:** The helper does not accept arbitrary HTML attributes such as `class`, `id`, or `title` for the label itself. Use surrounding CSS selectors or construct the markup directly when additional attributes are needed.
- **Dialog lifecycle:** Labels are normally created during InitializeScript() (p. 34), before the dialog's HTML document is generated.

Related functions

- Field Primitives (p. 93) - Create editable fields and bound display fields.
- AddLabelField (p. 99) - Create a Vectric-bound block-level label.
- AddLabelSpan() (p. 103) - Create a Vectric-bound inline label.
- AddElements() (p. 74) - Construct labels within nested element collections.
- AddDiv() (p. 80) - Group labels and controls into layout containers.
- AddStyle() (p. 108) - Apply CSS styling to labels and their surrounding elements.
- GetElementsHtml() (p. 351) - Generate HTML from structured element collections without registering bindings.

REFERENCE 4.6

Field Primitives

Gadget Factory provides three editable field primitives and one display-only field primitive that wrap Vectric's native `HTML_Dialog` field methods.

These functions simplify the creation of bound controls by generating the necessary HTML, registering the appropriate Vectric field bindings, and optionally connecting changes to Lua callbacks.

The editable primitives share a common calling convention, allowing programmers to use them interchangeably according to the required data type.

Syntax

Editable fields

```lua
GF.AddDoubleField(id, value, left, top, width, onchange)
GF.AddTextField(id, value, left, top, width, onchange)
GF.AddIntegerField(id, value, left, top, width, onchange)
```

Display-only field

```lua
GF.AddLabelField(id, value, left, top)
```

Return value: None for all four functions.

Common parameters

The three editable field primitives accept the following parameters:

Parameter	Description
`id`	Unique identifier for the HTML element and Vectric field binding.
`value`	Initial value of the field. Its expected type depends on the primitive.
`left`	Optional horizontal position in pixels.
`top`	Optional vertical position in pixels.
`width`	Optional field width in pixels.
`onchange`	Optional Lua function called when the field's value changes.

The `id` and `value` parameters are required for normal use. The remaining parameters are optional.

Basic usage

A field can be created with only its identifier and initial value:

```lua
GF.AddDoubleField("txtWidth", 12.5)
GF.AddTextField("txtName", "Untitled")
GF.AddIntegerField("txtCopies", 3)
```

Each call creates the appropriate HTML input and registers it with Vectric.

The programmer does not need to construct the HTML element or call the corresponding native `HTML_Dialog:Add...Field()` method separately.

Positioning and sizing

Fields normally participate in their containing HTML layout.

However, their positions can be specified explicitly:

```lua
GF.AddDoubleField("txtWidth", 12.5, 100, 50, 120)
```

This generates an HTML input with the following positioning attributes:

```css
style="position:absolute;left:100px;top:50px;width:120px;"
```

Both `left` and `top` must be supplied to enable absolute positioning. Supplying only one does not position the field.

Width is independent of positioning and may be specified without coordinates.

For example:

```lua
GF.AddTextField("txtName", "Untitled", nil, nil, 200)
```

This sets the field's width to 200 pixels while allowing the surrounding HTML and CSS to determine its position.

Omitting intermediate arguments

When using an editable field inside AddElements() (p. 74), optional arguments can be skipped using `false` placeholders.

```lua
GF.AddElements {
    {GF.AddDoubleField, "txtWidth", 12.5,
        false, false, 120, WidthChanged}
}
```

Here:

- `left` and `top` are omitted.
- `width` is 120 pixels.
- `WidthChanged` is the callback function.

Using `false` avoids creating gaps in the Lua array that could interfere with the positional arguments passed by `table.unpack()`.

This convention is particularly useful when a callback is required but explicit positioning is not.

Vectric bindings

Every field created by these primitives is registered with Vectric's native `HTML_Dialog` interface.

Gadget Factory handles two distinct operations:

1. Generate and collect the HTML representing the control.
2. Register a binding that adds the corresponding field to the dialog after its creation.

For example:

```lua
GF.AddTextField("txtName", "Untitled")
```

Generates an HTML input conceptually equivalent to:

```html
<input type="text" class="cssTXT" id="txtName">
```

It also registers a deferred call equivalent to:

```lua
Dlg:AddTextField("txtName", "Untitled")
```

The HTML input itself does not contain a `value` attribute. Its initial value is supplied through Vectric's field binding.

All three editable primitives generate HTML inputs with `type="text"`. Their behavior as text, integer, or floating-point fields is established through the corresponding native Vectric binding.

Field classes

Primitive	HTML class	Native binding
AddDoubleField	cssDBL	Dlg:AddDoubleField()
AddTextField	cssTXT	Dlg:AddTextField()
AddIntegerField	cssDBL	Dlg:AddIntegerField()

The classes allow Gadget Factory's shared CSS to style the different field types.

Change callbacks

All three editable primitives support an optional `onchange` function.

```lua
local WidthChanged = function(value)
    Width = value
end

GF.AddDoubleField(
    "txtWidth", Width,
    false, false, 120,
    WidthChanged
)
```

When the user commits a change to the field, Gadget Factory retrieves its current value and passes it to the specified callback.

The callback receives one argument: the value returned by the corresponding native Vectric field getter.

Primitive	Getter
AddDoubleField	Dlg:GetDoubleField()
AddTextField	Dlg:GetTextField()
AddIntegerField	Dlg:GetIntegerField()

Callbacks are supplied as function references, not function calls.

```lua
-- Correct
{GF.AddDoubleField, "txtWidth", Width,
    false, false, false, WidthChanged}

-- Incorrect
{GF.AddDoubleField, "txtWidth", Width,
    false, false, false, WidthChanged()}
```

The incorrect version executes `WidthChanged()` while constructing the element table, instead of passing the function for later use.

How callbacks are dispatched

Gadget Factory uses a hidden `LuaButton` to bridge the HTML input's `onchange` event to Lua.

When callback handling is required, the generated HTML contains an additional, hidden button with an identifier derived from the field ID.

For a field named `txtWidth`, the hidden button is named `chg_txtWidth`.

The visible input's HTML `onchange` handler clicks this hidden button. Vectric then invokes Gadget Factory's global LuaButton dispatcher, which retrieves the current field value and executes the registered callback.

This mechanism is handled automatically.

The programmer does not need to create the hidden button or register a separate handler.

The broader event-dispatching system is covered in AddHandler (p. 320).

Individual field types

AddDoubleField

```lua
GF.AddDoubleField(id, value, left, top, width, onchange)
```

Creates a Vectric floating-point numeric field.

```lua
GF.AddDoubleField("txtDepth", 0.125)
```

Distinctive behavior: automatic numeric formatting.

Gadget Factory uses an internal NumberFormatter (p. 281) configured to three decimal places for its DoubleFields.

The numeric value is formatted and converted back to a number before being supplied to Vectric.

For example, an initial value of `12.34567` is rounded to `12.346` before binding.

When the user changes a DoubleField, Gadget Factory:

1. Reads the entered value using `Dlg:GetDoubleField()`.
2. Rounds that value to three decimal places.
3. Updates the visible field using `Dlg:UpdateDoubleField()`.
4. Calls the programmer's `onchange` function, if supplied.

Important: The callback receives the value retrieved in step 1, before Gadget Factory's rounding operation.

Consequently, the value received by the callback may contain greater precision than the value subsequently displayed in the field.

Unlike TextFields and IntegerFields, DoubleFields automatically generate the hidden change-handling button even when no programmer-defined callback is supplied. This allows their rounding behavior to operate independently.

The three-decimal precision is fixed in the current factory implementation.

AddTextField

```lua
GF.AddTextField(id, value, left, top, width, onchange)
```

Creates a Vectric text field.

```lua
GF.AddTextField("txtDescription", "New Project")
```

Its initial value is supplied directly to `Dlg:AddTextField()` without numeric conversion or formatting.

If an `onchange` function is supplied, Gadget Factory retrieves the field contents using `Dlg:GetTextField()` and passes the result to the callback.

```lua
local DescriptionChanged = function(value)
    Description = value
end

GF.AddTextField(
    "txtDescription", Description,
    false, false, 200,
    DescriptionChanged
)
```

When no callback is supplied, Gadget Factory does not generate a hidden LuaButton for the TextField.

AddIntegerField

```lua
GF.AddIntegerField(id, value, left, top, width, onchange)
```

Creates a Vectric integer field.

```lua
GF.AddIntegerField("txtCopies", 3)
```

Like `AddDoubleField()`, it uses the `cssDBL` HTML class, but it is registered through Vectric's native `AddIntegerField()` method.

The distinction is therefore in the Vectric binding, not the HTML input type.

If a callback is supplied, Gadget Factory obtains the value using `Dlg:GetIntegerField()`.

Unlike DoubleFields, IntegerFields do not have an additional Gadget Factory formatting operation. Integer validation and interpretation are delegated to Vectric's native field implementation.

AddLabelField

`GF.AddLabelField()` creates a display-only label bound to Vectric's native label-field mechanism.

```lua
GF.AddLabelField(id, value, left, top)
```

Parameters

Parameter	Description
`id`	Unique identifier of the bound label.
`value`	Initial text displayed by the label.
`left`	Optional horizontal position in pixels.
`top`	Optional vertical position in pixels.

Example

```lua
GF.AddLabelField("labStatus", "Ready", 20, 200)
```

This generates the equivalent HTML:

```html
<div id="labStatus"
     style="position:absolute;left:20px;top:200px;">Ready</div>
```

The function registers the label using:

```lua
Dlg:AddLabelField("labStatus", "Ready")
```

A LabelField differs from the editable primitives in several respects:

- It generates an HTML `<div>` rather than an `<input>`.
- It does not accept a width argument.
- It does not accept an `onchange` callback.

- It does not allow direct user editing.
- Its contents can be updated programmatically after dialog creation.

Updating a LabelField

```lua
Dlg:UpdateLabelField("labStatus", "Processing...")
```

The existing HTML element remains in place while its displayed contents are updated.

AddLabelField versus AddLabelSpan

Both functions register the same underlying Vectric label binding.

Their principal difference is the HTML element they generate:

Primitive	Element	Default layout
`AddLabelField`	`<div>`	Block
`AddLabelSpan`	`<span>`	Inline

`AddLabelField()` also supports optional absolute positioning, while AddLabelSpan() (p. 103) is intended primarily for inline content.

Neither should be confused with AddControlLabel() (p. 86), which creates an HTML `<label>` associated with an input control rather than a Vectric-bound display field.

Example: Sectio Divina

Sectio Divina uses two DoubleFields to control its X and Y dimensions.

The following code is taken from its initialization routine:

```lua
-- numeric entry
local XChanged = function(v) DimensionChanged("X", v) end
local YChanged = function(v) DimensionChanged("Y", v) end

GF.AddElements {
    {GF.AddDiv, 'style="position:absolute;left:184px;top:150px;width:160px;"',
    {
        [[<h2 style="margin-bottom:6px;color:#214070;">Dimensions</h2>
        ]],
        {GF.AddDiv, 'class="cssTXTRow"', {
            {GF.AddControlLabel, "X:", "txtX"},
            {GF.AddDoubleField, "txtX", X, false, false, false, XChanged},
        }},
        {GF.AddDiv, 'class="cssTXTRow"', {
            {GF.AddControlLabel, "Y:", "txtY"},
            {GF.AddDoubleField, "txtY", Y, false, false, false, YChanged},
```

LUA / CONTINUED

```
        }},
    }}
}
```

This example illustrates several important conventions.

Layout: An outer `AddDiv()` establishes the dimensions panel's position. The two inner containers use the shared `cssTXTRow` class to arrange each label and its associated field.

Labels: `GF.AddControlLabel()` associates each descriptive label with its corresponding input ID. Its behavior is documented separately.

Positioning: The fields omit their individual coordinates and widths, leaving their layout to the containing HTML and CSS.

Callbacks: The `XChanged` and `YChanged` functions adapt the field's single-value callback to Sectio Divina's `DimensionChanged()` function, which also needs to know which dimension changed.

This last technique is particularly useful when several controls share the same underlying processing function.

Programmatically updating fields

Sectio Divina also provides a function for updating both numeric controls:

LUA

```lua
local SetDoubleFields = function(x, y)
    if x then Dlg:UpdateDoubleField("txtX", tonumber(D(x))) end
    if y then Dlg:UpdateDoubleField("txtY", tonumber(D(y))) end
end
```

Here, `D()` is the gadget's three-decimal numeric formatting function (p. 281).

`tonumber()` converts its formatted string back into a numeric value before passing it to Vectric.

This explicit formatting is necessary because Gadget Factory's automatic DoubleField formatting occurs during initial binding and user-initiated change handling; it does not intercept arbitrary calls to Vectric's native `UpdateDoubleField()` method.

Read-only HTML generators

Each field primitive also has a companion function that returns the corresponding HTML markup without appending it to the dialog or registering its native field binding.

LUA

```lua
GF.GetDoubleFieldHtml(id, value, left, top, width, onchange)
GF.GetTextFieldHtml(id, value, left, top, width, onchange)
GF.GetIntegerFieldHtml(id, value, left, top, width, onchange)
GF.GetLabelFieldHtml(id, value, left, top)
```

These functions support the read-only HTML generation mechanisms used by GetElementsHtml() (p. 351).

They generate markup only. In particular, they do not create working Vectric field bindings or register Lua callbacks.

Use the `Add...Field()` primitives for normal interactive controls.

Important considerations

- **Unique identifiers:** Every field must have a unique ID. Editable fields that use change handling also reserve an associated `chg_` identifier for their hidden LuaButton.
- **Type-appropriate initial values:** Supply numeric values to DoubleFields and IntegerFields, and strings to TextFields.
- **Optional positioning:** Both `left` and `top` must be supplied for absolute positioning. Width may be specified independently.
- **Callbacks:** The `onchange` argument must be a function reference when supplied.
- **Change timing:** The HTML `onchange` event normally occurs when a user commits an edit, rather than on every keystroke.
- **DoubleField precision:** DoubleFields automatically round values to three decimal places during initial binding and user-change handling.
- **Callback precision:** A DoubleField callback receives the value retrieved from Vectric before automatic display rounding.
- **Bound LabelFields:** These are not editable controls and do not generate change events.
- **No automatic HTML escaping:** Identifiers and initial label content must be suitable for their intended HTML context.
- **Dialog lifecycle:** Fields are normally created during InitializeScript() (p. 34). Native bindings are applied after Gadget Factory constructs the dialog.

Related functions

- AddElements() (p. 74) - Create fields as part of structured element collections.
- AddDiv() (p. 80) - Group fields into HTML containers.
- AddControlLabel() (p. 86) - Associate descriptive HTML labels with input fields.
- AddLabelSpan() (p. 103) - Create an inline Vectric-bound label.
- NumberFormatter (p. 281) - Configure reusable numeric formatting functions.
- AddHandler (p. 320) - Register callbacks through Gadget Factory's event-dispatch system.
- InitializeDialog (p. 49) - Access the instantiated Vectric dialog.

REFERENCE 4.7

AddLabelSpan()

`GF.AddLabelSpan()` creates an inline HTML `<span>` element and registers it as a Vectric label field.

Unlike ordinary HTML text, a bound label can be updated programmatically while the dialog is running, without regenerating its surrounding HTML.

This makes `AddLabelSpan()` particularly useful for displaying values that change during gadget execution, such as dimensions, calculated results, status messages, and other dynamic information.

Syntax

```lua
GF.AddLabelSpan(id, value)
```

Parameters

Parameter	Description
`id`	A unique identifier for the HTML element and its corresponding Vectric label binding.
`value`	The initial text displayed by the label. Normally supplied as a string.

Return value: None.

The function appends the generated HTML to Gadget Factory's content collection and registers the label for subsequent binding.

Basic usage

```lua
GF.AddLabelSpan("labStatus", "Ready")
```

This generates the following HTML:

```html
<span id="labStatus">Ready</span>
```

The `<span>` is an ordinary inline HTML element. It does not automatically create a new line or occupy the full width of its parent container.

For example:

```lua
GF.AddElements {
    "Status: ",
    {GF.AddLabelSpan, "labStatus", "Ready"}
}
```

Produces the equivalent HTML:

```html
Status: <span id="labStatus">Ready</span>
```

The text and label appear together in the normal flow of the document.

Vectric binding

The distinguishing feature of `AddLabelSpan()` is that it performs two operations:

1. Creates and appends the HTML `<span>` element.
2. Registers a binding that calls Vectric's `HTML_Dialog:AddLabelField()` method.

Conceptually, its implementation is equivalent to:

```lua
GF.AddLabelSpan = function(id, value)
    GF.AddElement('<span id="' .. id .. '">' .. value .. '</span>')

    GF.AddBinding(function(dlg)
        dlg:AddLabelField(id, value)
    end)
end
```

The binding is collected during interface construction and executed after Gadget Factory creates the dialog.

The HTML element's `id` connects the displayed content to its native Vectric label field.

The programmer does not need to call `dlg:AddLabelField()` separately. Gadget Factory handles the registration automatically.

Updating the label

Once the dialog has been created, its label can be updated using Vectric's native `UpdateLabelField()` method.

```lua
Dlg:UpdateLabelField("labStatus", "Processing...")
```

The displayed text changes without reconstructing the element or its surrounding HTML.

The identifier must match the one supplied to `GF.AddLabelSpan()`.

For example:

```lua
-- During InitializeScript
GF.AddLabelSpan("labStatus", "Ready")

-- Later, after dialog creation
GF.Dialog:UpdateLabelField("labStatus", "Complete")
```

Gadget Factory exposes the active dialog through `GF.Dialog`, allowing callback functions to interact with registered fields.

The label itself does not generate events or invoke callbacks. It is a display element whose value is changed programmatically.

Example: Sectio Divina

Sectio Divina uses `AddLabelSpan()` extensively to display its original, previous, and current dimensions.

The following excerpt is taken from its initialization routine (p. 34):

```lua
{GF.AddDiv, 'style="margin-top:20px;"', {
    [[<b style="margin-right:6px;">Current Dimensions:</b>]],
    {GF.AddLabelSpan, "labX", D(X)}, " × ",
    {GF.AddLabelSpan, "labY", D(Y)},
}},
```

This example combines several HTML construction techniques:

- `GF.AddDiv()` creates a container for the dimension display.
- A raw HTML string supplies the bold description.
- Two bound label spans display the X and Y dimensions.
- The multiplication symbol is inserted as ordinary text between the values.

The `D()` function was created using GF.NumberFormatter(3) (p. 281) and converts numeric values into formatted strings.

When the gadget changes the dimensions, it updates the corresponding labels:

```lua
Dlg:UpdateLabelField("labX", D(X))
Dlg:UpdateLabelField("labY", D(Y))
```

The surrounding description, multiplication symbol, and container remain unchanged.

This is a particularly convenient use of inline label fields: individual portions of a larger HTML expression can be updated independently.

Styling label spans

Because `AddLabelSpan()` generates an ordinary HTML `<span>` with an ID, its appearance can be controlled through CSS.

For example:

```lua
GF.AddStyle([[
    #labStatus {
        color: darkgreen;
        font-weight: bold;
    }
]])

GF.AddLabelSpan("labStatus", "Ready")
```

The span inherits ordinary CSS properties from its containing elements unless they are overridden.

Unlike AddLabelField (p. 99), `AddLabelSpan()` does not accept positioning arguments.

Its location is determined by the surrounding HTML layout, making it especially suitable for inline text and values embedded within sentences, headings, or other containers.

AddLabelSpan versus AddLabelField

Gadget Factory provides two related helpers for creating bound label elements.

Feature	AddLabelSpan	AddLabelField
Generated element	`<span>`	`<div>`
Default layout	Inline	Block
Vectric binding	`AddLabelField()`	`AddLabelField()`
Runtime updates	`UpdateLabelField()`	`UpdateLabelField()`
Position arguments	None	Optional Left and Top

Both functions use the same underlying Vectric binding mechanism. Their primary difference is the HTML element they generate and how that element participates in the page layout.

The separate AddControlLabel() (p. 86) function generates an HTML `<label>` associated with an input control. It does not register a Vectric label field and serves a different purpose.

GetLabelSpanHtml

Gadget Factory also exposes a read-only HTML generator:

```lua
GF.GetLabelSpanHtml(id, value)
```

It returns the corresponding HTML string without appending it to the dialog or registering a Vectric binding.

For example:

```lua
local html = GF.GetLabelSpanHtml("labStatus", "Ready")
```

The returned string is:

```html
<span id="labStatus">Ready</span>
```

This companion function is also used by GetElementsHtml() (p. 351) when rendering element collections without executing their normal registration operations.

Use `AddLabelSpan()` when the label must be updated through Vectric. Generating the HTML alone does not establish the native label binding.

Important considerations

- **Unique IDs:** Each bound label must have a unique identifier within the dialog.
- **Initial values:** Supply a string containing the desired initial text. Use `tostring()` or NumberFormatter (p. 281) when converting numeric values for display.
- **No automatic HTML escaping:** The supplied value is inserted directly into the HTML. Any markup it contains will be interpreted by the HTML parser.
- **No automatic positioning:** The generated span follows normal inline layout rules.
- **Binding timing:** The Vectric binding is registered during interface construction but applied only after the dialog object exists.
- **Runtime updates:** Use `UpdateLabelField()` after dialog creation. Calling `AddLabelSpan()` again is not a substitute for updating an existing bound field.
- **No event handler:** Label spans display information but do not themselves provide an `onchange` callback.

Related functions

- AddElement() (p. 72) - Append raw HTML.
- AddElements() (p. 74) - Process structured element collections.
- AddDiv() (p. 80) - Create HTML containers.
- AddLabelField (p. 93) - Create a bound block-level label with optional positioning.
- AddControlLabel() (p. 86) - Create an HTML label associated with an input control.
- GetElementsHtml() (p. 351) - Generate HTML without registering bindings.
- NumberFormatter (p. 281) - Format numeric values for display.

REFERENCE 4.8

AddStyle()

`GF.AddStyle()` adds CSS styling rules to the gadget's generated HTML document.

It allows programmers to define interface styling directly in Lua without creating or maintaining a separate CSS file.

The supplied CSS is collected during initialization and inserted into a `<style>` element in the HTML document's `<head>` when Gadget Factory constructs the dialog.

Syntax

```lua
GF.AddStyle(style)
```

Parameters

Parameter	Description
`style`	A string containing one or more CSS rules.

Return value: None.

Basic usage

```lua
GF.AddStyle("body {background-color: #f7f0d9;}")
```

This produces a CSS rule within the document's `<head>`:

```html
<style>
body {background-color: #f7f0d9;}
</style>
```

The programmer supplies CSS declarations only. `AddStyle()` generates the surrounding `<style>` element automatically when the document is assembled.

Multiple CSS rules

Lua's multiline strings are particularly convenient for writing CSS:

```lua
GF.AddStyle([[
    .dimensions {
        position: absolute;
        left: 50px;
        top: 25px;
        width: 350px;
    }

    .dimensions h3 {
        color: #214070;
        margin-bottom: 6px;
    }
]])
```

CSS may use any selectors and properties supported by the embedded browser engine.

This approach keeps interface construction and its associated styling together in the gadget's Lua source.

Multiple AddStyle calls

`AddStyle()` can be called repeatedly.

```lua
GF.AddStyle([[
    body {
        background-color: #f7f0d9;
    }
]])

GF.AddStyle([[
    .dimensions {
        font-weight: bold;
    }
]])
```

All supplied CSS is collected into a single `<style>` element.

Important: Gadget Factory concatenates the supplied strings without inserting additional whitespace or line breaks. Each fragment should therefore contain whatever separation is needed between adjacent CSS rules.

CSS declarations follow the browser's normal cascading rules. When selectors have equal specificity, later declarations generally take precedence over earlier ones.

Example: Sectio Divina

Sectio Divina uses `AddStyle()` to control several aspects of its interface without requiring an additional stylesheet.

Background color

```lua
-- aureic background
GF.AddStyle("body {background-color: #f7f0d9;}")
```

This gives the dialog its pale golden background, complementing the gadget's use of the golden ratio.

Button styling

```lua
GF.AddStyle([[
.sdButtons #ButtonCancel,
.sdButtons #ButtonOK,
.sdButtons input[type="button"] { width:100px; height:35px; }]])
```

This rule sizes buttons inside the `.sdButtons` container.

Rather than modifying individual button definitions, the programmer applies a shared CSS rule to the relevant elements.

Dynamic visibility

Sectio Divina also uses CSS to control the visibility of its manual recalculation controls:

```lua
GF.AddStyle([[
.cssToggle input[type="checkbox"]:checked ~ .manualRecalc { display:none; }]])
```

When the checkbox is checked, the `.manualRecalc` element is hidden.

This behavior is handled entirely by CSS, without requiring Lua to update the element's visibility.

It demonstrates how CSS can manage simple visual states independently of Gadget Factory's event-handling system.

Stylesheet integration

Gadget Factory builds the document's styling in the following order:

1. Links to external stylesheets, beginning with `Gadgets.css` and including stylesheets registered by loaded components.
2. CSS supplied through `GF.AddStyle()`.
3. An optional body-background rule generated from `GF.GadgetDef.Background`.

The collected inline CSS appears after the external stylesheet links, allowing gadget-specific rules to override shared styles where normal CSS cascade rules permit.

If `GF.GadgetDef.Background` is specified, Gadget Factory appends a background-color rule using `!important`. This takes precedence over an ordinary body background declaration supplied through `AddStyle()`.

When to use AddStyle

Use `AddStyle()` for styling that belongs specifically to the current gadget, including:

- Dialog-specific positioning and layout.
- Custom colors, fonts, and spacing.
- Styling for manually constructed HTML.
- Overrides or extensions of shared component styles.
- CSS-driven visual states.

For reusable styling shared across multiple gadgets, a dedicated stylesheet associated with the appropriate component library is generally preferable.

Execution behavior

When called, `AddStyle()` appends its argument to Gadget Factory's internal CSS collection.

The CSS is not immediately applied to the dialog.

When Gadget Factory constructs the HTML document, it combines the collected CSS fragments and places them in the document's `<head>`.

Consequently, `AddStyle()` should normally be called during InitializeScript() (p. 34), before the dialog is created.

Its position relative to `AddElement()`, `AddElements()`, and `AddDiv()` calls does not determine where the CSS appears in the final document.

Important considerations

- **CSS only:** Do not include `<style>` tags in the supplied string.
- **Multiple calls:** CSS fragments are concatenated without automatic separators.
- **No validation:** Gadget Factory does not parse, sanitize, or validate the supplied CSS.
- **CSS precedence:** Normal cascading, specificity, and inheritance rules apply.
- **Browser compatibility:** CSS features must be supported by Vectric's embedded browser engine.
- **No native binding:** `AddStyle()` does not create Vectric fields or register callbacks.
- **Initialization:** Calling `AddStyle()` after dialog creation does not update the existing document's stylesheet.

CHAPTER 05

05

Components

Reusable controls for choices, actions, lists, and editable suggestions.

IN THIS CHAPTER

Components: overview 113

RadioGroup 120

CheckboxGroup 137

ButtonGroup 156

ToolStrip 173

Combobox 192

ListBox 217

DataPicker 246

LayerPicker 252

FileList 257

REFERENCE 5.1

Components: overview

Gadget Factory provides a collection of reusable interface components that extend Vectric's native `HTML_Dialog` capabilities.

These components simplify the construction of interactive interfaces by combining HTML generation, CSS styling, native field bindings, event handling, and application-specific behavior behind relatively small programming interfaces.

Some components provide convenient wrappers around familiar controls. Others implement functionality not directly available through Vectric's native dialog API.

All components are constructed and configured using ordinary Lua code, leaving the programmer in control of the interface and its behavior.

Component categories

The component library is organized into three general categories: **Groups**, **Lists**, and **Pickers**.

Groups

Groups organize related controls or actions within a common interface region.

Component	Purpose
RadioGroup (p. 120)	Select one option from a collection of mutually exclusive choices.
CheckboxGroup (p. 137)	Manage multiple independent Boolean options.
ButtonGroup (p. 156)	Arrange action buttons in a configurable grid.
ToolStrip (p. 173)	Present a compact collection of action buttons docked to an interface edge.

These components eliminate much of the repetitive HTML and callback registration normally associated with constructing groups of controls.

Lists

Lists present collections of items and manage their selection.

Component	Purpose
Combobox (p. 192)	Provide a compact dropdown with dynamically editable choices.
ListBox (p. 217)	Provide a scrollable, single-selection list with dynamic item management.

Component	Purpose
FileList (p. 257)	Display files from a directory, with integrated descriptive information and editing facilities.

Combobox provides an enhanced alternative to Vectric's native dropdown control.

ListBox constructs a conventional-looking selection interface from individually styled HTML buttons.

FileList is a more specialized component that combines directory enumeration, file selection, and persistent descriptive metadata.

Despite their different implementations, all three present straightforward interfaces for retrieving the user's selection.

Pickers

Pickers combine editable text with suggested values. The user may select a suggestion or type a new value.

Component	Purpose
DataPicker (p. 246)	Edit text with an application-supplied list of suggestions.
LayerPicker (p. 252)	Choose an existing job layer name or enter a new one, excluding system-layer names.

Both provide an optional `OnValidityChanged(valid)` callback. It reports Boolean validity transitions rather than every text edit. External native text updates (p. 250) require an explicit refresh when validity notification is needed.

Loading components

Component libraries are loaded through GF.LoadComponents() (p. 36).

For example:

```lua
GF.LoadComponents {"RadioGroup", "CheckboxGroup", "ListBox"}
```

This makes the named components available through the `GF` table.

Each component library is loaded once, regardless of how many instances the gadget subsequently creates.

Component stylesheets are registered automatically. The programmer does not need to add their CSS files manually.

The LoadComponents (p. 36) chapter documents library loading, source and compiled modules, duplicate-load handling, and public API exposure.

Common construction conventions

Most components share a familiar two-stage construction pattern:

```lua
local Component = GF.ComponentName.New(...)
    :WithContent(...)
```

The constructor creates the instance and establishes its configuration.

`WithContent()` supplies the items, layout, or other content needed to complete the component, then registers it with Gadget Factory.

For example:

```lua
local ModeGroup = GF.RadioGroup.New("Mode")
    :WithContent({
        {"Standard", "standard"},
        {"Fine",     "fine"}
    }, 20, 50, 250)
    :Select(1)
```

The result is a RadioGroup with two choices, positioned at the supplied coordinates and initially selecting Standard.

Lua's dot and colon conventions

Notice the two different method-call operators.

```lua
GF.RadioGroup.New("Mode")
```

uses dot notation because `New()` is a constructor belonging to the component class.

```lua
ModeGroup:Select(2)
```

uses colon notation because `Select()` operates on a particular instance.

In Lua, the colon automatically supplies the instance as the first argument, conventionally named `self`.

This is why the component's instance methods use the colon syntax.

Fluent construction

Many component methods return the instance on which they operate.

This allows calls to be chained:

```lua
GF.RadioGroup.New("Mode")
    :WithContent(items, 20, 50, 250)
    :Select(1)
```

Chaining is optional.

The same operations can be written separately:

```lua
local ModeGroup = GF.RadioGroup.New("Mode")
ModeGroup:WithContent(items, 20, 50, 250)
ModeGroup:Select(1)
```

Both forms are valid. The programmer may choose whichever is clearer in the surrounding code.

Programmer-defined values

Selection components generally distinguish between the text displayed to the user and the value returned to Lua.

For example:

```lua
{"Fine Finish", 0.025}
```

The displayed label is `Fine Finish`, while the associated Lua value is `0.025`.

The value may be a number, string, or another supported Lua type, depending on the component.

This eliminates the need to translate interface indices or displayed labels into meaningful application values throughout the gadget.

FileList is an exception: its selected value is a table containing file information rather than an arbitrary programmer-defined value.

DataPicker and LayerPicker return the exact edited string through `GetText()`. Their suggestions do not pair labels with separate programmer-defined values.

Events and callbacks

Components use Gadget Factory's centralized event-dispatching system (p. 320) to execute programmer-defined Lua callbacks.

For example:

```lua
local ModeChanged = function(value, group)
    CurrentMode = value
end
```

LUA / CONTINUED

```lua
local ModeGroup = GF.RadioGroup.New("Mode", ModeChanged)
    :WithContent({
        {"Standard", "standard"},
        {"Fine",     "fine"}
    }, 20, 50, 250)
```

The programmer supplies a function reference, and RadioGroup handles the remaining event-registration details.

Callback signatures differ according to the component's purpose.

Selection groups generally provide the selected value and originating instance. CheckboxGroup additionally supplies the key identifying the changed checkbox.

ButtonGroup and ToolStrip execute individual button actions with their optional arguments.

The corresponding reference chapters describe these conventions in detail.

Programmatic changes do not ordinarily invoke the component's selection callback. When changing a selection must also trigger application logic, that logic should be called explicitly.

The pickers have a separate validity-transition contract (p. 249). Their refresh methods can notify when current validity changes. An external `UpdateTextField()` does not itself supply the observed editing events; refresh the picker to synchronize notification state.

Preparing and rendering components

The component library also follows a useful architectural convention.

Most components expose three related operations:

Method	Responsibility
`PrepareContent()`	Prepare the component's content and internal representation.
`GetHtml()`	Generate its HTML representation.
`WithContent()`	Prepare, render, and register the working component.

For normal gadget development, `WithContent()` is the appropriate choice.

The separate preparation and rendering methods are useful for more specialized applications, particularly visual development tools that need to examine or display component HTML without registering native bindings.

Forge itself makes extensive use of this separation.

These methods do not have identical side-effect profiles across every component. For example, FileList's preparation (p. 271) reads directory contents and associated details from the filesystem. Combobox (p. 206) and ListBox (p. 233) can refresh their visible item collections if the instance is already live.

Use a separate, uninstalled instance when preparing HTML for a preview.

Consult the individual component chapters before using these facilities independently.

Component styling

Component layout and appearance are defined by companion CSS stylesheets.

LayerPicker shares DataPicker's rendering and stylesheet rather than defining another editable control.

Gadget Factory automatically incorporates these stylesheets when the corresponding libraries are loaded.

Gadget-specific overrides can be supplied through AddStyle (p. 108).

This allows programmers to modify a component's appearance without changing its implementation.

However, Vectric's embedded Trident browser imposes some limitations on HTML and CSS behavior. The component stylesheets are designed with those constraints in mind.

For ordinary use, the default styling is generally sufficient.

Construction and dialog lifecycle

Components are normally created during InitializeScript (p. 34), before Gadget Factory constructs the native dialog.

After the HTML document has been generated, Gadget Factory registers the components' native bindings and establishes their dialog references.

Once the dialog is running, the programmer can retrieve selections, update values, or invoke other supported instance methods.

Some components provide additional runtime capabilities, such as changing the contents of a ListBox or Combobox.

Others, such as FileList, establish their collections during initialization and do not support runtime item modification.

Understanding these differences is important when choosing a component.

Choosing the appropriate component

The component offering the most functionality is not necessarily the best choice.

For a small number of mutually exclusive options, RadioGroup may be clearer than a dropdown.

When several independent settings are required, CheckboxGroup avoids constructing and managing individual checkboxes.

ButtonGroup and ToolStrip offer different ways to organize actions, depending on the available space and desired layout.

For item collections, the choice between Combobox, ListBox, and FileList depends on whether the contents are dynamic, how much information should remain visible, and whether filesystem integration is required.

Each component is intended to solve a particular class of interface problems without restricting the programmer's ability to construct alternatives using Gadget Factory's HTML primitives.

Summary

Gadget Factory's component library provides reusable controls that hide much of the repetitive HTML, binding, and event-handling work involved in Vectric gadget development.

The components share consistent construction conventions while retaining APIs suited to their individual responsibilities.

Their internal implementations range from straightforward native-control wrappers to complete custom interfaces assembled from ordinary HTML, CSS, JavaScript, and Lua.

The programmer describes the controls and their purpose. Gadget Factory handles the machinery that makes them work.

When no existing component meets a requirement, the facilities described in Advanced Topics (p. 316) provide the tools necessary to construct something new.

REFERENCE 5.2

RadioGroup

`GF.RadioGroup` creates a group of mutually exclusive radio buttons, allowing the user to select one option from a collection of predefined choices.

Each option consists of a descriptive label and an associated Lua value. Gadget Factory handles the HTML construction, native Vectric field binding, initial selection, and optional callback registration.

RadioGroup supports multiple independent instances within the same dialog and uses the same general construction conventions as Gadget Factory's other group components.

Loading the component

RadioGroup is supplied as a separate library module and must be loaded before use.

```lua
GF.LoadComponents {"RadioGroup"}
```

This makes the `GF.RadioGroup` constructor available and includes the component's associated stylesheet.

The module is loaded once per gadget. Multiple RadioGroup instances may then be created without additional library loads.

Basic usage

A RadioGroup is created using `New()`, populated using `WithContent()`, and optionally given an initial selection using `Select()`.

```lua
local OrientationGroup = GF.RadioGroup.New("Orientation")
    :WithContent({
        {"Portrait",  1},
        {"Landscape", 2}
    }, 30, 140, 300)
    :Select(2)
```

This creates a group titled **Orientation**, containing two radio buttons.

The group is positioned using the specified Left, Top, and Width values.

The second option, Landscape, is initially selected.

The values `1` and `2` are programmer-defined Lua values associated with the respective choices.

Construction sequence

The normal sequence consists of three operations:

1. `New()` creates the RadioGroup instance and establishes its optional callback.

2. `WithContent()` defines the available choices, constructs the HTML, and registers the component with Gadget Factory.
3. `Select()` establishes an initial selection, if something other than the first item is desired.

All three methods support the fluent construction style shown above.

New

`RadioGroup.New()` creates a new RadioGroup instance.

Syntax

```lua
local group = GF.RadioGroup.New(title, onselect)
```

Parameters

Parameter	Description
`title`	Optional heading displayed above the radio buttons.
`onselect`	Optional Lua function invoked when the user selects an item.

Return value: A new RadioGroup instance.

Creating a group without a callback

```lua
local Options = GF.RadioGroup.New("Options")
```

The group can be populated and its selected value retrieved later without requiring an event handler.

Creating a group with a callback

```lua
local SelectionChanged = function(value, group)
    -- Respond to the selected value
end

local Options = GF.RadioGroup.New("Options", SelectionChanged)
```

The callback receives two arguments:

Argument	Description
`value`	The Lua value associated with the selected item.
`group`	The RadioGroup instance that generated the event.

The second argument is useful when a shared callback handles several groups or needs access to the originating instance.

Optional title

If `title` is omitted or `nil`, the component does not generate a heading.

When supplied, the title is enclosed in an HTML `<h2>` element within the group.

WithContent

`WithContent()` defines the group's choices, layout, and callback registrations.

Syntax

```lua
group:WithContent(items, left, top, width)
```

Parameters

Parameter	Description
`items`	Array-style Lua table containing the available choices.
`left`	Horizontal position in pixels.
`top`	Vertical position in pixels.
`width`	Width of the group in pixels.

Return value: The RadioGroup instance.

Item structure

Each item is a two-element Lua table:

```lua
{label, value}
```

The first element is the text displayed beside the radio button.

The second is the Lua value associated with that choice.

For example:

```lua
local Options = GF.RadioGroup.New("Cut Direction")
    :WithContent({
        {"Climb",        "CLIMB"},
        {"Conventional", "CONVENTIONAL"}
    }, 20, 50, 250)
```

The displayed labels and returned values are independent.

Selecting **Conventional** produces the value `"CONVENTIONAL"`, not the displayed label.

Values are not restricted to integers

RadioGroup stores the supplied values in an ordinary Lua table.

For example:

```lua
local DepthGroup = GF.RadioGroup.New("Pass Depth")
    :WithContent({
        {"Shallow", 0.02},
        {"Medium",  0.03},
        {"Deep",    0.05}
    }, 20, 50, 250)
```

The selected value can be used directly in calculations without translating a radio-button index into a numeric depth.

String values work equally well.

Because the implementation uses a sequential array of values, each item should have a non-`nil` value. Omitting values can create gaps that interfere with Lua's array-length and indexing behavior.

Registration

`WithContent()` performs more than HTML construction.

It prepares the item collection, generates the group's markup, registers the component for native Vectric binding, and establishes individual event handlers when an `onselect` callback was supplied to `New()`.

Consequently, `WithContent()` is normally called **once per RadioGroup instance**, during InitializeScript (p. 34).

It is not intended as a general-purpose runtime method for replacing the choices in an already registered group.

Select

`Select()` establishes or changes the selected radio button by its position in the item collection.

Syntax

```lua
group:Select(index)
```

Parameters

Parameter	Description
`index`	One-based index of the item to select.

Return value: The RadioGroup instance.

Initial selection

```lua
local Options = GF.RadioGroup.New("Options")
    :WithContent({
        {"First",  10},
        {"Second", 20},
        {"Third",  30}
    }, 20, 50, 250)
    :Select(2)
```

The second item is selected.

Notice that the argument to `Select()` is **the item's index, not its associated value**.

In this example:

```lua
Options:Select(2)
```

selects the item whose value is `20`.

Passing `20` would not select that item unless the group actually contained at least twenty entries.

Programmatic selection

Once the dialog has been created and the RadioGroup bound to Vectric, `Select()` can also change the displayed selection.

```lua
Options:Select(3)
```

Internally, the component calls:

```lua
dlg:UpdateRadioIndex(id, index)
```

This keeps the visible selection synchronized with the requested index.

`Select()` does not explicitly invoke the programmer's `onselect` callback. It is intended to establish or update the selection, not execute the associated application logic.

Invalid indices

The implementation ignores indices outside the valid range.

For example:

```lua
Options:Select(0)
Options:Select(99)
```

Neither call changes the selection in a three-item group.

The method returns the RadioGroup instance without raising an error for an out-of-range numeric index.

Important: `Select(0)` is not a supported method of clearing the selection.

When the native binding is created without a previously established valid selection, RadioGroup supplies index `1` to Vectric.

Thus, the first item is the default.

Selected

`Selected()` retrieves the current selection from the native Vectric dialog.

Syntax

```lua
local value, index = group:Selected()
```

Return values

Return	Description
`value`	Lua value associated with the selected item, or `nil` if no valid item is selected.
`index`	One-based index reported by Vectric, or `0` when no item is selected.

Example

Given:

```lua
local Options = GF.RadioGroup.New("Options")
    :WithContent({
        {"First",  10},
        {"Second", 20},
        {"Third",  30}
    }, 20, 50, 250)
    :Select(2)
```

After the dialog has been initialized:

```lua
local value, index = Options:Selected()

-- value = 20
-- index = 2
```

The method obtains the index using Vectric's native:

```lua
dlg:GetRadioIndex(id)
```

It then uses that index to retrieve the associated Lua value.

Retrieving only the value

Lua permits a function returning multiple values to be assigned to a single variable.

```lua
local value = Options:Selected()
```

Only the first returned value is retained.

This is convenient when the item's associated value is all the application needs.

Retrieving only the index

```lua
local _, index = Options:Selected()
```

The underscore is an ordinary Lua variable commonly used by convention for values that will be ignored.

When Selected can be called

`Selected()` requires the component to have been bound to the native dialog.

It should therefore be used after dialog creation, typically from a callback or another appropriate runtime operation.

Calling it during `InitializeScript()` is premature because the RadioGroup has not yet received its `HTML_Dialog` reference.

If the desired initial index is already known during initialization, retain that value in the gadget's own application variables or use the corresponding argument to `Select()`.

Selection callbacks

RadioGroup's optional `onselect` function provides immediate notification when the user selects an item.

Consider:

```lua
local PassDepth

local DepthChanged = function(value, group)
    PassDepth = value
end

local DepthGroup = GF.RadioGroup.New("Pass Depth", DepthChanged)
    :WithContent({
        {"0.02", 0.02},
        {"0.03", 0.03},
        {"0.05", 0.05}
    }, 20, 50, 250)
```

When the user selects the third option, RadioGroup invokes:

```lua
DepthChanged(0.05, DepthGroup)
```

The callback receives the **associated value**, rather than requiring the programmer to retrieve or translate the selected index.

The second argument identifies the originating RadioGroup instance.

Sharing a callback

Several groups may use the same callback function.

```lua
local GroupChanged = function(value, group)
    if group == DepthGroup then
        PassDepth = value
    elseif group == DirectionGroup then
        CutDirection = value
    end
end
```

The instance argument makes it possible to distinguish which group generated the event.

Groups without callbacks

The `onselect` argument is optional.

Without it, RadioGroup does not generate LuaButton callback handling for its individual radio inputs.

The group still receives a native Vectric radio-group binding, and its selection remains accessible through `Selected()`.

This is appropriate when the selection is needed only after the user has finished interacting with the dialog.

Example: Sectio Divina

Sectio Divina uses RadioGroup to select the orientation of its dimensional transformation.

The following excerpt comes from its initialization routine:

```lua
OrientationGroup = GF.RadioGroup.New("Orientation", OrientationChanged)
    :WithContent({
        {"Portrait",  1},
        {"Landscape", 2}
    }, 30, 141, 308)
    :Select(Orientation)
```

The example demonstrates several important conventions.

Component construction: `New()` creates the instance with a heading and selection callback.

Item values: Portrait and Landscape are associated with numeric values `1` and `2`.

Layout: `WithContent()` supplies the group's position and width.

Initial selection: `Select()` uses the current value of `Orientation` as an item index.

In this particular gadget, the associated values happen to match the item indices. That is convenient but not required by RadioGroup.

Responding to orientation changes

Sectio Divina defines the callback as follows:

```lua
local OrientationChanged = function(value)
    if value == Orientation then return end
    Orientation = value
    if not Dlg:GetCheckBox("togAuto") then return end
    X, Y = Y, X
    SetDoubleFields(X, Y)
end
```

The function receives the selected orientation value.

If the selection has not changed, it returns immediately.

Otherwise, it updates the application variable and, when automatic recalculation is enabled, exchanges the X and Y dimensions before updating their corresponding fields.

Although RadioGroup supplies the originating instance as a second callback argument, this particular function does not need it.

Lua permits functions to ignore arguments they do not declare.

The example demonstrates how RadioGroup can communicate user selections directly to application logic without exposing the details of Vectric's radio-button indices.

Preparing and rendering content independently

RadioGroup exposes two additional methods that separate content preparation and HTML generation from ordinary component registration.

These methods are particularly useful to visual designers and other tools that need to render a component without registering it as an active dialog control.

PrepareContent

```lua
group:PrepareContent(items, left, top, width)
```

Accepts the same item collection and layout arguments as `WithContent()`.

Return value: The RadioGroup instance.

It stores the layout values, prepares the individual item definitions, and builds the table mapping item indices to their associated Lua values.

It does not register the component with Gadget Factory or establish its event handlers.

GetHtml

```lua
local html = group:GetHtml()
```

Return value: A string containing the RadioGroup's generated HTML.

The function uses the content and layout previously established by `PrepareContent()`.

It does not append the HTML to Gadget Factory's dialog content collection or register native Vectric bindings.

Example

```lua
local PreviewGroup = GF.RadioGroup.New("Preview")

PreviewGroup:PrepareContent({
    {"Small",  1},
    {"Medium", 2},
    {"Large",  3}
}, 20, 50, 250)

local html = PreviewGroup:GetHtml()
```

The returned string contains the complete radio-group markup.

This is useful when a tool needs to inspect or display the HTML representation independently of the normal dialog construction process.

The returned HTML does not, by itself, create a working Vectric-bound RadioGroup.

For normal gadget interfaces, `WithContent()` remains the preferred method.

How RadioGroup works internally

RadioGroup is implemented as a Lua object using a shared metatable and a per-instance state table.

Each call to `New()` creates an independent instance with its own title, values, selection state, and optional callback.

Automatic identifiers

RadioGroup generates its own identifiers using Gadget Factory's component utilities.

For example, the first instance uses a group ID equivalent to:

```lua
rbg1
```

Its first radio input receives an ID equivalent to:

```lua
rgi1_rbg1
```

The outer HTML container receives the ID:

```lua
divrbg1
```

Individual radio inputs share the group name `rbg1`, allowing the browser and Vectric to treat them as one mutually exclusive selection.

Programmers do not normally need to construct or manage these identifiers.

Generated HTML

A simplified two-item group produces markup resembling:

```html
<div id="divrbg1" style="left:30px;top:141px;width:308px;">
    <div class="cssRBG">
        <h2>Orientation</h2>

        <div class="cssRB">
            <input type="radio" id="rgi1_rbg1" name="rbg1">
            <label for="rgi1_rbg1">Portrait</label>
        </div>

        <div class="cssRB">
            <input type="radio" id="rgi2_rbg1" name="rbg1">
            <label for="rgi2_rbg1">Landscape</label>
```

HTML / CONTINUED

```
        </div>
    </div>
</div>
```

When an `onselect` callback is configured, the radio inputs additionally receive the `LuaButton` class.

The labels use ordinary HTML `for` attributes to associate their text with the corresponding radio inputs.

The component's stylesheet controls the visual layout and appearance.

Native binding

RadioGroup registers itself with Gadget Factory when `WithContent()` executes.

After the native dialog is created, Gadget Factory invokes its `Bind()` method.

The implementation is:

LUA

```lua
function RadioGroup:Bind(dlg)
    self.dlg = dlg
    dlg:AddRadioGroup(self.id, self.selected or 1)
end
```

The component retains the dialog reference and registers the entire group using Vectric's native `AddRadioGroup()` method.

The initial selection is supplied as an index, defaulting to `1` when none has been established.

Event registration

When a callback is supplied, `WithContent()` registers a handler for each radio input.

The essential logic is:

LUA

```lua
local ItemClick = function(idx)
    self.onSelection(self.values[idx], self)
end

for i, Item in ipairs(self.content) do
    Core.AddHandler(Item.id, ItemClick, i)
end
```

Each radio input receives the same Lua callback function, but with a different item index as its registered argument.

When a button is activated, Gadget Factory's central dispatcher supplies that index to `ItemClick()`.

The function retrieves the corresponding Lua value and calls the programmer's callback, supplying both the selected value and originating instance.

This is the same dispatch mechanism documented in Event Handling (p. 320).

Why retain the values separately?

Vectric's native RadioGroup binding deals with the selected item index.

RadioGroup maintains its own table mapping those indices to programmer-defined values.

This allows the application to work with meaningful values such as `"CLIMB"`, `"CONVENTIONAL"`, or `0.025`, rather than interpreting raw radio-button positions throughout the gadget.

The mapping is simple, but it makes the public interface considerably more convenient.

Styling

The accompanying `RadioGroup.css` stylesheet controls the appearance of the group's container, heading, radio buttons, and descriptive labels.

The stylesheet is registered automatically when RadioGroup is loaded:

```lua
GF.LoadComponents {"RadioGroup"}
```

No additional CSS configuration is required for normal use.

Group container

The main radio group uses the `cssRBG` class.

```css
.cssRBG {
    position: absolute;
    width: 100%;
    padding: 6px 10px 15px 18px;
    box-sizing: border-box;
    border: 1px solid #AAA;
}
```

The component generates an outer positioning container using the Left, Top, and Width arguments supplied to `WithContent()`.

The inner `cssRBG` container occupies the full available width and provides padding and a border around the group.

Its height is determined by its contents.

Group heading

When a title is supplied to `New()`, RadioGroup generates an HTML `<h2>` element.

Its appearance is controlled by:

```css
.cssRBG h2 {
    margin: 2px 0 12px 0;
    color: #214070;
}
```

The heading uses a dark-blue color and includes additional spacing beneath it to separate the title from the selectable items.

Radio button rows

Each radio button and its descriptive label are enclosed in a `cssRB` container.

```css
.cssRBG .cssRB {
    position: relative;
    height: 20px;
}
```

Adjacent rows receive additional vertical spacing:

```css
.cssRBG .cssRB + .cssRB {
    margin-top: 12px;
}
```

The + operator is the CSS *adjacent sibling combinator*. It selects a radio row only when it immediately follows another radio row.

This avoids applying the extra spacing before the first item.

Radio button appearance

Individual radio inputs use:

```css
.cssRB input[type='radio'] {
    position: absolute;
    left: 12px;
    top: 0;
    width: 20px;
    height: 20px;
    margin: 0;
    padding: 0;
}
```

The radio button is positioned within its row and given a consistent 20 × 20 pixel size.

RadioGroup also uses the Microsoft-specific `::-ms-check` pseudo-element to customize the appearance of the radio control in Vectric's embedded Trident browser.

For example, the selected state uses a blue-toned radial gradient:

```css
.cssRB input[type='radio']:checked::-ms-check {
    color: transparent;
    background-image:
        radial-gradient(circle at 40% 35%,
        #8EFFFF 0%,
        #247D9C 25%,
        #191970 65%,
        #101050 100%);
}
```

These rules alter the native radio button's presentation without affecting its underlying selection behavior.

Because `::-ms-check` is browser-specific, the precise appearance may differ when the HTML is displayed outside Vectric's Trident environment.

Radio labels

Each radio button has an associated HTML `<label>`.

```css
.cssRB label {
    position: absolute;
    left: 40px;
    top: -1px;
    height: 20px;

    color: black;
    font-size: 16px;
    line-height: 20px;
}
```

The label is positioned beside its radio button, with sufficient horizontal separation to avoid overlapping the control.

The generated HTML uses the label's `for` attribute to associate it with the corresponding radio input.

This also allows the label text to serve as a clickable selection target through standard HTML behavior.

Gadget-specific styling

A gadget can override RadioGroup's appearance using AddStyle (p. 108).

For example:

```lua
GF.AddStyle([[
    .cssRBG {
        border-color: #8ca2b0;
    }

    .cssRBG h2 {
        color: #2b4f63;
    }
```

LUA / CONTINUED

```
    .cssRBG .cssRB + .cssRB {
        margin-top: 8px;
    }

    .cssRB label {
        font-size: 14px;
    }
]])
```

These rules change the group's border and heading colors, reduce the spacing between rows, and use smaller descriptive labels.

The component's HTML and native Vectric bindings remain unchanged.

Unlike ButtonGroup (p. 156), RadioGroup does not accept an additional CSS class name through its constructor.

Consequently, the example above affects all RadioGroup instances matching those selectors. To style one instance independently, use a more specific selector targeting its outer container.

When overriding the appearance of the actual radio buttons, remember that their custom styling relies on Trident-specific CSS.

Presentation belongs to CSS; selection values, native bindings, and callbacks remain the responsibility of RadioGroup.

Important considerations

- **Library loading:** Load `RadioGroup` through `GF.LoadComponents()` before creating an instance.
- **Multiple instances:** Each RadioGroup is independent and receives automatically generated identifiers.
- **Item structure:** Each item should contain a label and a non-`nil` Lua value.
- **One-based indexing:** `Select()` accepts an item index, not the associated value.
- **Default selection:** The first item is selected by default when no valid index has been established.
- **Invalid selection:** Out-of-range numeric indices are ignored. `Select(0)` does not clear the selection.
- **Runtime selection:** `Select()` updates the native selection after binding but does not explicitly execute the selection callback.
- **Native state:** `Selected()` reads the current selection from Vectric rather than relying solely on the instance's stored initial index.
- **Callback arguments:** The selection callback receives the associated value and the originating RadioGroup instance.
- **Optional callback:** A group without a callback still supports normal native selection and value retrieval.
- **Construction timing:** Call `WithContent()` during interface initialization, before Gadget Factory creates the native dialog.
- **Single registration:** Calling `WithContent()` repeatedly on the same registered instance can cause duplicate event-handler registrations and duplicate content.

- **Read-only rendering:** `PrepareContent()` and `GetHtml()` generate markup without registering native bindings.
- **HTML content:** Titles and item labels are inserted directly into HTML without automatic escaping.
- **CSS styling:** Layout and visual appearance depend on the component's stylesheet and the surrounding dialog styles.

Related functions

- LoadComponents (p. 36) - Load Gadget Factory component libraries.
- CheckboxGroup (p. 137) - Create a group of independently selectable checkboxes.
- ButtonGroup (p. 156) - Create a collection of action buttons.
- Event Handling (p. 320) - Understand the centralized LuaButton callback system.
- AddBinding (p. 329) - Understand native field registration.
- InitializeScript (p. 34) - Construct and register components.
- InitializeDialog (p. 49) - Access native dialog functionality after binding.

Summary

`GF.RadioGroup` provides a compact interface for creating mutually exclusive selections from programmer-defined choices.

Its principal methods are:

Method	Purpose
`New()`	Create a RadioGroup instance.
`WithContent()`	Define its options, generate HTML, and register the component.
`Select()`	Set the selected item by index.
`Selected()`	Retrieve the selected Lua value and index.
`PrepareContent()`	Prepare values and layout without registration.
`GetHtml()`	Generate the group's HTML independently.

The component handles the HTML construction, native binding, automatic identifiers, and optional callback dispatching.

The programmer supplies the choices and decides what their values mean.

RadioGroup manages the selection. The gadget manages what that selection represents.

REFERENCE 5.3

CheckboxGroup

`GF.CheckboxGroup` creates a collection of independently selectable checkboxes, allowing the user to enable or disable multiple options within a single interface component.

Each checkbox has a unique programmer-defined key, a descriptive label, and an initial Boolean value.

Unlike RadioGroup (p. 120), which permits only one selection at a time, CheckboxGroup supports any combination of checked and unchecked items.

The component provides flexible methods for retrieving and changing individual values, selected subsets, or the entire collection.

Loading the component

CheckboxGroup is supplied as a separate library module and must be loaded before use.

```lua
GF.LoadComponents {"CheckboxGroup"}
```

This exposes the `GF.CheckboxGroup` constructor and includes its associated stylesheet.

The module is loaded once per gadget. Any number of CheckboxGroup instances may then be created.

Basic usage

A CheckboxGroup is created using `New()` and populated using `WithContent()`.

```lua
local Options = GF.CheckboxGroup.New("Surfacing Options")
    :WithContent({
        {"FinalPass",   "Final Pass",   true},
        {"CreateGroup", "Create Group", false},
        {"CreateSheet", "New Sheet",    false}
    }, 20, 50, 250)
```

This creates a group titled **Surfacing Options**, containing three checkboxes.

The first option is initially checked, while the remaining two are unchecked.

Each entry contains three values:

```lua
{key, label, checked}
```

The key identifies the option programmatically, while the label is the text displayed beside its checkbox.

CheckboxGroup manages the HTML, native Vectric bindings, and optional event handling automatically.

New

`CheckboxGroup.New()` creates a new CheckboxGroup instance.

Syntax

```lua
local group = GF.CheckboxGroup.New(title, onselect)
```

Parameters

Parameter	Description
`title`	Optional heading displayed above the collection.
`onselect`	Optional Lua callback invoked when the user changes a checkbox.

Return value: A new CheckboxGroup instance.

Without a callback

```lua
local Options = GF.CheckboxGroup.New("Options")
```

The group operates normally, but changes do not invoke programmer-defined Lua code.

This is appropriate when the values will be retrieved only when the gadget is ready to process its results.

With a callback

```lua
local OptionChanged = function(key, value, group)
    -- Respond to the changed option
end

local Options = GF.CheckboxGroup.New("Options", OptionChanged)
```

The callback receives three arguments:

Argument	Description
`key`	Programmer-defined key of the checkbox that changed.
`value`	Current Boolean value of that checkbox.
`group`	CheckboxGroup instance that generated the event.

The instance argument is particularly useful when multiple groups share the same callback.

WithContent

`WithContent()` defines the group's items, layout, and optional event registrations.

Syntax

```lua
group:WithContent(items, left, top, width)
```

Parameters

Parameter	Description
`items`	Array-style table containing checkbox definitions.
`left`	Horizontal position in pixels.
`top`	Vertical position in pixels.
`width`	Optional width in pixels.

Return value: The CheckboxGroup instance.

The returned instance allows the constructor and content definition to be chained.

Item structure

Each item uses the following convention:

```lua
{key, label, checked}
```

Element	Description
`key`	Unique identifier used when reading or changing the checkbox value.
`label`	Descriptive text displayed beside the checkbox.
`checked`	Optional initial Boolean value. Defaults to `false`.

For example:

```lua
local Features = GF.CheckboxGroup.New("Features")
    :WithContent({
        {"Preview", "Generate Preview", true},
        {"Outline", "Include Outline",  false},
        {"Labels",  "Show Labels",      true}
    }, 30, 70, 240)
```

The keys are independent of their displayed labels.

This means the programmer can change the label text without changing the identifier used throughout the application.

Initial values

The third item value establishes the checkbox's initial checked state.

```lua
{"Preview", "Generate Preview", true}
```

If omitted, the checkbox is initially unchecked:

```lua
{"Preview", "Generate Preview"}
```

Internally, CheckboxGroup normalizes initial values using:

```lua
local checked = not not Item[3]
```

This is a common Lua technique for converting a value to a Boolean.

Remember that Lua considers only `false` and `nil` to be false. Values such as `0` and `""` are considered true.

Consequently, explicitly supplying `true` or `false` is preferable.

Optional width

The `width` argument may be omitted:

```lua
group:WithContent(items, 30, 70)
```

In this case, the outer container is generated without an explicit width declaration.

When supplied, the width is inserted into the container's inline CSS.

The `left` and `top` arguments specify its location.

Registration

`WithContent()` performs three principal operations:

1. Prepares the checkbox definitions and their internal key-to-ID mapping.
2. Registers individual event handlers when an `onselect` callback is configured.
3. Generates the HTML and registers the component with Gadget Factory for native binding.

It should normally be called once per instance during InitializeScript (p. 34).

The component's native Vectric checkboxes are registered later, when Gadget Factory constructs the dialog.

GetValues

`GetValues()` retrieves the current checked states from Vectric.

It provides several calling conventions, allowing the programmer to retrieve one value, multiple values, a selected subset, or the complete collection.

Rather than implementing separate functions for these operations, CheckboxGroup examines the number and types of arguments supplied.

This technique is sometimes called *simulated function overloading*.

Syntax

```lua
group:GetValues(...)
```

Supported forms

Calling convention	Return value
`GetValues()`	Table containing all keys and Boolean values
`GetValues(key)`	Boolean for one key
`GetValues(key1, key2, ...)`	Multiple Boolean return values
`GetValues({key1, key2, ...})`	Table containing the requested keys and values

Retrieve all values

```lua
local values = Options:GetValues()
```

The returned table is keyed by the option identifiers:

```lua
{
    FinalPass   = true,
    CreateGroup = false,
    CreateSheet = false
}
```

Individual values can then be accessed using ordinary Lua table syntax.

```lua
if values.FinalPass then
    -- Include the finishing pass
end
```

Important: The no-argument form returns the component's internal values table, not an independent snapshot.

If the programmer retains this table and later calls `GetValues()` or `SetValues()`, its contents may change.

To preserve a historical snapshot, copy the values into a separate table.

Retrieve one value

```lua
local finalPass = Options:GetValues("FinalPass")
```

The result is a Boolean.

This is the simplest approach when the application needs only one option.

Retrieve multiple values

```lua
local finalPass, createSheet =
    Options:GetValues("FinalPass", "CreateSheet")
```

The returned values correspond to the supplied keys in the same order.

This is particularly convenient when several Boolean variables must be updated together.

Retrieve a selected subset as a table

```lua
local values = Options:GetValues({
    "FinalPass",
    "CreateSheet"
})
```

The result is a keyed table:

```lua
{
    FinalPass   = true,
    CreateSheet = false
}
```

Unlike the no-argument form, this table is newly constructed for the requested subset.

It can therefore be retained without subsequent CheckboxGroup updates modifying that particular table.

How GetValues works

Regardless of which calling convention is used, CheckboxGroup obtains the requested states through Vectric's native checkbox getter:

```lua
self.dlg:GetCheckBox(id)
```

The component maps the supplied programmer-defined key to its internally generated HTML ID.

For each requested item, it reads the native value, updates its internal state table, and returns the result.

This ensures that `GetValues()` reflects the current dialog state rather than relying solely on the initial checkbox values.

A useful Lua technique

The implementation distinguishes calling conventions using:

```lua
local n = select("#", ...)
```

In Lua, `...` represents the variable argument list supplied to a function.

`select("#", ...)` returns the number of arguments, allowing the function to distinguish between calls containing no arguments, one argument, or several.

It then examines the argument types to determine whether the programmer supplied individual keys or a table.

When multiple values are requested, the function uses:

```lua
return table.unpack(values, 1, n)
```

This allows one function to return several independent Boolean results.

The mechanism is simple, but it produces a remarkably flexible public interface.

SetValues

`SetValues()` changes checkbox states programmatically.

Like `GetValues()`, it supports several calling conventions to accommodate common operations without requiring separate setter functions.

Syntax

```lua
group:SetValues(...)
```

Return value: None.

Supported forms

Calling convention	Behavior
`SetValues(true)`	Check all items
`SetValues(false)`	Uncheck all items
`SetValues(boolean, key1, ...)`	Set named items to the same Boolean
`SetValues({key = boolean, ...})`	Set named items from a keyed table
`SetValues(key1, boolean1, ...)`	Set individual keys to corresponding values

Set every checkbox

```lua
Options:SetValues(true)
```

Checks every item in the group.

```lua
Options:SetValues(false)
```

Unchecks every item.

This is useful for Reset, Select All, and Clear All operations.

Set several checkboxes to the same value

```lua
Options:SetValues(true, "FinalPass", "CreateGroup")
```

Checks only the two named options.

Similarly:

```lua
Options:SetValues(false, "FinalPass", "CreateSheet")
```

Unchecks the specified items without modifying the remaining checkboxes.

Notice that the Boolean comes first in this calling convention.

Set values from a table

```lua
Options:SetValues({
    FinalPass   = true,
    CreateGroup = false,
    CreateSheet = true
})
```

The function updates each specified item using its corresponding Boolean value.

Items omitted from the table are left unchanged.

This approach is particularly convenient when restoring previously saved settings or applying a collection of related options.

Set values using key-value pairs

```lua
Options:SetValues(
    "FinalPass", true,
    "CreateGroup", false,
    "CreateSheet", true
)
```

Each key is followed by the value to assign.

This form avoids constructing a separate table when only a few values must be updated.

How SetValues works

Every calling convention ultimately uses the same internal operation:

```lua
local function SetValue(key, value)
    value = not not value
    self.values[key] = value
    self.dlg:UpdateCheckBox(self.items[key], value)
end
```

The function performs three steps:

1. Converts the supplied value to a Boolean.
2. Updates the component's internal values table.
3. Updates the corresponding native Vectric checkbox.

This keeps the component's stored values synchronized with the native dialog after programmatic changes.

Programmatic updates and callbacks

`SetValues()` does not explicitly invoke the group's `onselect` callback.

This is an important distinction from a user-generated checkbox event.

```lua
Options:SetValues(true, "FinalPass")
```

changes the checkbox state, but the application should not depend on its selection callback executing as a result.

If changing a value programmatically must also trigger application logic, call that logic explicitly.

This keeps programmatic state changes separate from user interaction.

Argument considerations

The key-value calling convention expects complete pairs:

```lua
Options:SetValues("FinalPass", true)
```

An incomplete pair is not rejected by the current implementation. A missing value becomes `nil`, which is converted to `false`.

For predictable behavior, always provide both members of every pair.

Unknown keys are not validated before being passed to Vectric's checkbox methods. Programmers should therefore use keys defined in the original item collection.

Selection callbacks

The optional `onselect` function is invoked when the user interacts with a checkbox configured for LuaButton event handling.

Consider:

```lua
local OptionChanged = function(key, checked, group)
    if key == "FinalPass" then
        -- Respond to the new state
    end
end
```

Construct the group with this callback:

```lua
local Options = GF.CheckboxGroup.New("Options", OptionChanged)
    :WithContent({
        {"FinalPass", "Final Pass", true},
        {"Preview",   "Preview",    false}
    }, 20, 50, 250)
```

When the user changes Final Pass, CheckboxGroup:

1. Receives the event through Gadget Factory's LuaButton dispatcher.

2. Resolves the affected checkbox using its key.
3. Reads its current value from Vectric.
4. Updates the component's internal values table.
5. Invokes the programmer's callback.

The callback receives the key, current Boolean value, and originating group instance.

Sharing a callback

Several CheckboxGroups may share one function.

```lua
local OptionsChanged = function(key, checked, group)
    if group == SurfacingOptions then
        -- Handle surfacing settings
    elseif group == OutputOptions then
        -- Handle output settings
    end
end
```

The originating instance distinguishes otherwise similar controls.

Without a callback

If no callback is supplied to `New()`, the generated checkboxes do not use the `LuaButton` class.

They remain ordinary native-bound Vectric checkboxes.

The programmer can retrieve their current values through `GetValues()` whenever necessary.

This can eliminate unnecessary callback activity for options that do not affect the interface while the dialog is open.

Example: Surfacing options

A surfacing gadget provides a useful example of CheckboxGroup's flexible API.

The following illustrative example uses options similar to those in our Surfacer project.

```lua
local SurfacingOptions

local function InitializeScript(job)

    GF.LoadComponents {"CheckboxGroup"}

    SurfacingOptions = GF.CheckboxGroup.New("Options")
        :WithContent({
            {"FinalPass",   "Finishing Pass", true},
            {"CreateGroup", "Group Toolpaths", false},
            {"CreateSheet", "Create New Sheet", false}
        }, 20, 80, 250)
end
```

Because no selection callback is supplied, the group simply collects the user's choices.

When the gadget is ready to process its results, it can retrieve the entire collection:

```lua
local options = SurfacingOptions:GetValues()

if options.FinalPass then
    -- Add the finishing pass
end

if options.CreateGroup then
    -- Group the generated toolpaths
end

if options.CreateSheet then
    -- Create a new sheet
end
```

Alternatively, it can retrieve just the values it needs:

```lua
local finalPass, newSheet =
    SurfacingOptions:GetValues("FinalPass", "CreateSheet")
```

These examples demonstrate why programmer-defined keys are so useful.

The application works directly with meaningful option names rather than indexing an anonymous collection of checkbox values.

Restoring saved options

Suppose the gadget has previously saved a collection of settings:

```lua
local SavedOptions = {
    FinalPass   = true,
    CreateGroup = true,
    CreateSheet = false
}
```

Once the dialog has been created and the component bound to Vectric, the settings can be restored using:

```lua
SurfacingOptions:SetValues(SavedOptions)
```

This applies the stored Boolean values to their corresponding native checkboxes.

For initial settings known before dialog creation, supplying the values directly in `WithContent()` is simpler.

Preparing and rendering content independently

Like RadioGroup (p. 120), CheckboxGroup provides separate content-preparation and HTML-rendering methods.

These are especially useful for visual designers, previews, and other tools that need a component's HTML without registering it as an active dialog control.

PrepareContent

```lua
group:PrepareContent(items, left, top, width)
```

Accepts the same arguments as `WithContent()`.

Return value: The CheckboxGroup instance.

The method:

1. Stores the group's position and optional width.
2. Creates an internal representation of each item.
3. Generates automatic HTML IDs.
4. Builds mappings between programmer-defined keys and those IDs.
5. Stores the initial Boolean states.

It does not register event handlers or add the component to Gadget Factory's binding collection.

Calling it again replaces the instance's prepared item definitions and state.

It should therefore not be used casually to reconfigure an already bound component.

GetHtml

```lua
local html = group:GetHtml()
```

Return value: A string containing the generated CheckboxGroup HTML.

`GetHtml()` uses the definitions previously established by `PrepareContent()`.

It does not register native bindings or callbacks, and it does not append the generated HTML to Gadget Factory's document collection.

Example

```lua
local Preview = GF.CheckboxGroup.New("Preview")

Preview:PrepareContent({
    {"Grid",   "Show Grid", true},
    {"Labels", "Show Labels", false}
}, 20, 50, 240)

local html = Preview:GetHtml()
```

The result is a complete HTML representation of the group.

This can be used by a design tool that needs to display or inspect the component's markup.

The generated HTML alone does not establish a working Vectric-bound CheckboxGroup.

For ordinary gadget development, `WithContent()` remains the appropriate method.

Relationship to WithContent

The current implementation begins with:

```lua
function CheckboxGroup:WithContent(items,left,top,width)
    self:PrepareContent(items,left,top,width)
```

It subsequently registers any requested event handlers and adds the generated HTML and component binding through Gadget Factory's internal facilities.

This provides a useful architectural separation:

PrepareContent defines the component. GetHtml renders it. WithContent registers it.

The pattern is shared by several of Gadget Factory's reusable components.

How CheckboxGroup works internally

CheckboxGroup uses a Lua table and shared metatable to represent each independent component instance.

Every instance maintains its own identifier, title, item definitions, current values, and optional callback.

Automatic identifiers

Each instance receives an automatically generated group identifier.

For example, the first group uses:

```lua
cbg1
```

The first checkbox in that group receives an identifier equivalent to:

```lua
chk1_cbg1
```

The outer container receives:

```lua
divcbg1
```

These identifiers are generated through Gadget Factory's shared component utilities.

The programmer normally works with the supplied keys rather than these internal HTML IDs.

Generated HTML

A simplified two-item group generates markup equivalent to:

```html
<div id="divcbg1" style="left:20px;top:50px;width:240px;">
    <div class="cssCBG">
        <h2>Options</h2>
        <label class="cssCHK">
            <input type="checkbox" id="chk1_cbg1" checked>
            Preview
        </label>
        <label class="cssCHK">
            <input type="checkbox" id="chk2_cbg1">
            Create Sheet
        </label>
    </div>
</div>
```

The actual HTML is generated from a series of string templates.

CheckboxGroup places each checkbox inside an HTML `<label>`, making the descriptive text part of the control's clickable area.

This is another standard HTML technique that eliminates the need for a separate AddControlLabel (p. 86) call.

When an `onselect` callback is configured, the generated checkbox inputs also receive:

```css
class="LuaButton"
```

This allows Vectric to report user interactions through Gadget Factory's central event dispatcher.

Native binding

When Gadget Factory creates the dialog, it invokes the component's `Bind()` method.

The implementation is:

```lua
function CheckboxGroup:Bind(dlg)
    for key,id in pairs(self.items) do
        dlg:AddCheckBox(id,self.values[key])
    end
    self.dlg = dlg
end
```

Each checkbox is registered individually using Vectric's native `AddCheckBox()` method.

Its initial state comes from the values prepared during `WithContent()`.

The component also retains the dialog reference for subsequent calls to `GetValues()` and `SetValues()`.

This is why those methods require a bound dialog rather than merely a constructed CheckboxGroup instance.

Event dispatch

When the programmer supplies a callback, CheckboxGroup registers an event handler for each checkbox.

The relevant logic is:

```lua
local ItemClick = function(key)
    local id = self.items[key]
    local value = self.dlg:GetCheckBox(id)

    self.values[key] = value
    self.onSelection(key, value, self)
end
```

The component's handler reads the current native value and forwards the result to the programmer's callback.

The associated key is supplied through Gadget Factory's existing event-dispatch system.

This follows the same general design as RadioGroup, but with an important distinction: CheckboxGroup operates on named Boolean states rather than a single selected index.

Styling

The supplied `CheckboxGroup.css` stylesheet defines the appearance of individual checkbox rows.

The `.cssCHK` class uses a flex layout to align the checkbox and its descriptive text.

```css
.cssCHK {
    display: flex;
    align-items: center;
    height: 32px;

    color: #212529;
    font-size: 16px;
}
```

Individual checkbox inputs are sized to 22 × 22 pixels, with a small margin separating the control from its label text.

The stylesheet also uses the Microsoft-specific `::-ms-check` pseudo-element to customize the checkbox's appearance in Trident.

For example, checked boxes receive a blue-toned radial-gradient background.

These rules are part of the component's presentation and are loaded automatically with its stylesheet.

The programmer can add gadget-specific overrides using AddStyle (p. 108) without modifying the component library.

CheckboxGroup versus RadioGroup

Both components use similar construction conventions, but they serve different selection models.

Feature	CheckboxGroup	RadioGroup
Selection model	Multiple independent values	One selected item
Item definition	`{key, label, checked}`	`{label, value}`
Value type	Boolean per item	Programmer-defined value
Main getter	`GetValues()`	`Selected()`
Programmatic setter	`SetValues()`	`Select()`
Callback	`(key, value, group)`	`(value, group)`
Native Vectric binding	Individual checkboxes	Radio group

The difference in item structure is particularly important.

RadioGroup identifies a selected item by its position and returns an associated value.

CheckboxGroup identifies each option by a stable key and maintains a separate Boolean value for each.

Important considerations

- **Library loading:** Load `CheckboxGroup` before constructing an instance.
- **Multiple instances:** Each group receives its own automatically generated IDs and state.
- **Unique keys:** Keys must be unique within a group. Duplicate keys can cause multiple HTML inputs to map to the same stored entry.
- **Stable keys:** Use consistent, meaningful identifiers because the keys are the public interface for retrieving and changing values.
- **Initial values:** The third item entry defaults to `false` when omitted.

- **Boolean conversion:** Lua values are converted using `not not`. Zero and empty strings are true in Lua.
- **Optional callback:** Without an `onselect` callback, checkboxes remain bound but do not generate Gadget Factory selection events.
- **Callback arguments:** The callback receives the changed key, current Boolean state, and originating group instance.
- **Programmatic changes:** `SetValues()` does not explicitly invoke the selection callback.
- **Native state:** `GetValues()` reads the current values from Vectric and updates the component's internal state.
- **Internal table reference:** The no-argument `GetValues()` form returns the component's own values table, not a copy.
- **Unknown keys:** The getter and setter do not validate requested keys before using them with Vectric's field methods.
- **Runtime timing:** `GetValues()` and `SetValues()` require the component to have been bound to an existing dialog.
- **Complete key-value pairs:** The variable-argument setter should always receive complete pairs.
- **Single registration:** `WithContent()` should normally be called once per instance.
- **No setter chaining:** Unlike `WithContent()`, `SetValues()` does not return the instance.
- **Read-only rendering:** `PrepareContent()` and `GetHtml()` produce HTML without registering native fields or event handlers.
- **HTML content:** Titles and item labels are inserted directly into HTML without automatic escaping.
- **CSS compatibility:** The component's appearance relies partly on Trident-specific checkbox styling.

Related functions

- LoadComponents (p. 36) - Load Gadget Factory component libraries.
- RadioGroup (p. 120) - Create mutually exclusive selections.
- ButtonGroup (p. 156) - Create grouped action buttons.
- Event Handling (p. 320) - Understand the centralized callback dispatcher.
- AddBinding (p. 329) - Understand native Vectric field registration.
- AddControlLabel (p. 86) - Create ordinary HTML labels associated with controls.
- AddStyle (p. 108) - Apply gadget-specific CSS.
- InitializeScript (p. 34) - Construct and register interface components.

Summary

`GF.CheckboxGroup` provides a flexible interface for managing multiple independent Boolean options.

Its public methods are:

Method	Purpose
`New()`	Create a CheckboxGroup instance.
`WithContent()`	Define items, generate HTML, and register the component.
`GetValues()`	Retrieve one, several, or all checkbox values.
`SetValues()`	Update selected checkboxes or the entire group.
`PrepareContent()`	Prepare the component without registration.
`GetHtml()`	Generate its HTML independently.

The component combines keyed Boolean values, native Vectric bindings, optional callbacks, and convenient bulk operations.

RadioGroup answers “Which one?” CheckboxGroup answers “Which ones?”

The programmer decides what those choices mean.

REFERENCE 5.4

ButtonGroup

`GF.ButtonGroup` creates a collection of HTML action buttons arranged in a rectangular grid.

Each button can invoke a Lua callback, optionally receive a programmer-defined argument, and display a tooltip explaining its purpose.

ButtonGroup also supports buttons with explicitly assigned HTML identifiers, including Vectric's special `ButtonOK` and `ButtonCancel` identifiers used for dialog completion.

Unlike RadioGroup (p. 120) and CheckboxGroup (p. 137), ButtonGroup does not maintain a selection or register native Vectric fields. Its primary responsibilities are **button layout, HTML generation, and event registration**.

Loading the component

ButtonGroup is supplied as a separate library module and must be loaded before use.

```lua
GF.LoadComponents {"ButtonGroup"}
```

This exposes the `GF.ButtonGroup` constructor and includes the associated `ButtonGroup.css` stylesheet.

The library is loaded once per gadget. Multiple ButtonGroup instances can then be created independently.

Basic usage

A ButtonGroup is created using `New()` and populated using `WithContent()`.

```lua
local Actions = GF.ButtonGroup.New(1, 3)
    :WithContent({
        {"Preview", RefreshPreview},
        {"Reset",   ResetSettings},
        {"Help",    ShowHelp}
    }, 20, 50)
```

This creates a horizontal group containing three buttons.

Each button invokes its corresponding Lua function when activated.

The arguments `1, 3` specify one row and three columns, while `20, 50` specify the group's position in pixels.

Construction sequence

The normal sequence consists of two operations:

1. `New()` creates the ButtonGroup instance and defines its grid dimensions and optional styling class.

2. `WithContent()` prepares the buttons, registers their callbacks, generates the HTML, and adds it to Gadget Factory's dialog content.

Both methods return the instance, allowing the fluent construction style shown above.

New

`ButtonGroup.New()` creates a new ButtonGroup instance.

Syntax

```lua
local group = GF.ButtonGroup.New(rows, cols, style)
```

Parameters

Parameter	Description
`rows`	Number of rows in the button grid.
`cols`	Number of columns in the button grid.
`style`	Optional additional CSS class name for the group.

Return value: A new ButtonGroup instance.

Grid dimensions

For a horizontal arrangement of four buttons:

```lua
local Actions = GF.ButtonGroup.New(1, 4)
```

For a vertical arrangement:

```lua
local Actions = GF.ButtonGroup.New(4, 1)
```

A grid containing six buttons in two columns and three rows can be created using:

```lua
local Actions = GF.ButtonGroup.New(3, 2)
```

The `rows` and `cols` arguments determine the available positions, not necessarily the number of buttons actually generated.

An item must be supplied for each position that should contain a button.

Additional styling

The optional `style` argument appends another CSS class to the group's normal `cssBTNG` class.

For example:

```lua
local Actions = GF.ButtonGroup.New(1, 3, "compactActions")
```

The inner container receives:

```html
<div class="cssBTNG compactActions">
```

This allows the programmer to introduce custom styling without changing the component's stylesheet.

The `style` parameter expects a CSS class name, not an inline CSS declaration.

An example of customizing the appearance appears later in this chapter.

WithContent

`WithContent()` defines the buttons, their actions, and the group's position.

Syntax

```lua
group:WithContent(items, left, top)
```

Parameters

Parameter	Description
`items`	Array-style table containing button definitions.
`left`	Horizontal position in pixels.
`top`	Vertical position in pixels.

Return value: The ButtonGroup instance.

Unlike the selection-group components, ButtonGroup does not accept a width argument in `WithContent()`.

Its dimensions are determined by the number of rows and columns, individual button sizes, and CSS spacing.

Button definition

Each item may contain four positional values:

```lua
{label, action, args, title}
```

Element	Description
`label`	Text displayed on the button.
`action`	Lua function or explicit HTML button identifier.
`args`	Optional argument passed to a Lua callback.
`title`	Optional HTML tooltip displayed when hovering over the button.

Only the label and action are normally required.

For example:

```lua
{"Apply", ApplyChanges}
```

Creates a button labeled Apply that invokes `ApplyChanges()`.

Supplying an argument

```lua
{"Calculate X", Recalculate, "X"}
{"Calculate Y", Recalculate, "Y"}
```

Both buttons invoke the same function but supply different arguments.

Supplying a tooltip

```lua
{"Apply", ApplyChanges, nil, "Apply the current settings"}
```

The fourth item becomes the HTML `title` attribute.

The third entry is `nil` because this button does not require a callback argument.

ButtonGroup explicitly retrieves positions 1 through 4 using `table.unpack(Item, 1, 4)`, so this optional argument gap is supported.

The resulting button contains:

```css
title="Apply the current settings"
```

The tooltip is supplied by the browser. ButtonGroup does not implement separate tooltip behavior.

Button actions

The `action` entry determines how ButtonGroup constructs and registers an individual button.

There are two distinct behaviors.

Lua function actions

When `action` is a function, ButtonGroup generates a unique HTML identifier, assigns the required `LuaButton` class, and registers the callback with Gadget Factory.

For example:

```lua
{"Refresh", RefreshPreview}
```

Creates HTML resembling:

```html
<input type="button"
       class="LuaButton"
       id="btn1_btng1"
       value="Refresh">
```

The generated ID will depend on the ButtonGroup instance and the item's position.

The associated Lua function is registered using Gadget Factory's event-dispatching mechanism.

When the user activates the button, Vectric reports the event to Gadget Factory, which executes the registered callback.

See Event Handling (p. 320) for a detailed explanation.

Explicit identifier actions

When `action` is not a function, ButtonGroup uses that value directly as the button's HTML ID.

For example:

```lua
{"Commit", "ButtonOK"}
{"Discard", "ButtonCancel"}
```

This produces button elements with the specified identifiers.

Unlike function actions, these buttons do not receive automatically generated IDs or Gadget Factory handler registrations.

The current implementation omits the `LuaButton` class for string actions. Buttons using `ButtonOK` or `ButtonCancel` use Vectric's native completion mechanism; they do not register an ordinary ButtonGroup callback. Raw HTML buttons with these IDs worked without `LuaButton` in the Hello-Millie Aspire test (p. 289).

The intended use is with Vectric's special dialog-completion identifiers. An arbitrary string is not interpreted as a Lua function name.

For example:

```lua
{"Refresh", "RefreshPreview"}
```

does not register or invoke a Lua function named `RefreshPreview`.

Use a function reference when a button must execute programmer-defined Lua code.

Passing arguments to callbacks

ButtonGroup allows a single callback function to serve several buttons.

```lua
local Recalculate = function(axis)
    if axis == "X" then
        CalculateX()
    elseif axis == "Y" then
        CalculateY()
    end
end
```

The group can then use:

```lua
GF.ButtonGroup.New(1, 2)
    :WithContent({
        {"Calculate X", Recalculate, "X"},
        {"Calculate Y", Recalculate, "Y"}
    }, 20, 50)
```

When the first button is activated, Gadget Factory effectively invokes:

```lua
Recalculate("X")
```

The second button invokes:

```lua
Recalculate("Y")
```

Multiple values in one argument

ButtonGroup supplies the `args` entry to AddHandler (p. 320).

That mechanism passes exactly one argument to the callback.

If several values are required, they can be packaged into a table:

```lua
local Process = function(options)
    Calculate(options.axis, options.precision)
end

GF.ButtonGroup.New(1, 2)
    :WithContent({
        {"Fine X", Process, {
            axis = "X",
            precision = 0.001
        }},

        {"Coarse Y", Process, {
            axis = "Y",
            precision = 0.01
        }}
    }, 20, 50)
```

The callback receives the table as a single argument.

Unlike RadioGroup and CheckboxGroup, **ButtonGroup does not automatically pass its own instance to the programmer's callback**.

If that instance or other application state is needed, the programmer can make it accessible through an enclosing Lua function or a suitably scoped variable.

Grid layout

ButtonGroup supports both row-oriented and column-oriented layouts.

The choice is determined automatically by the relationship between `rows` and `cols`.

Row-oriented layout

When the number of rows is less than or equal to the number of columns, ButtonGroup generates an HTML container for each row.

For example:

```lua
GF.ButtonGroup.New(2, 3)
```

With six items, the intended arrangement is:

```text
[ Button 1 ] [ Button 2 ] [ Button 3 ]
[ Button 4 ] [ Button 5 ] [ Button 6 ]
```

Each row is represented by a container using the `cssButtonRow` class.

Column-oriented layout

When the number of rows exceeds the number of columns, ButtonGroup generates an HTML container for each column.

For example:

```lua
GF.ButtonGroup.New(3, 2)
```

The intended arrangement is:

```text
[ Button 1 ] [ Button 2 ]
[ Button 3 ] [ Button 4 ]
[ Button 5 ] [ Button 6 ]
```

However, the underlying HTML is organized into two columns rather than three rows.

Each column receives the `cssButtonColumn` class.

Why two different layouts?

The component uses CSS to arrange buttons within HTML row or column containers.

The decision is simple:

```lua
if self.rows > self.cols then
    -- Generate columns
else
    -- Generate rows
end
```

This supports horizontal and vertical arrangements without requiring separate components or an explicit layout-mode parameter.

Item ordering

Items are indexed in normal row-major order, regardless of the generated container arrangement.

For a three-row, two-column group:

Position	Column 1	Column 2
Row 1	Item 1	Item 2
Row 2	Item 3	Item 4
Row 3	Item 5	Item 6

Internally, ButtonGroup uses:

```lua
(row - 1) * self.cols + col
```

to calculate each item's index.

In the column-oriented case, it traverses those indices column by column when generating HTML.

This allows the programmer to supply one consistent item-array ordering for both layout styles.

Missing items

The number of supplied items does not have to equal `rows * cols`.

If an item is missing, ButtonGroup skips the corresponding button.

For example, a two-row, three-column grid containing only four items will generate four buttons, but it does not reserve empty cells for the missing entries. The remaining buttons flow together, and shorter rows are centered by the component's CSS.

Items beyond the configured grid capacity are ignored.

For predictable alignment, supply a complete rectangular collection. If the design needs intentional gaps, use an explicitly arranged HTML/CSS layout.

Example: Sectio Divina

Sectio Divina makes extensive use of ButtonGroup because its interface requires more actions than the standard Execute and Cancel pair can provide.

Its button group contains five vertically arranged actions.

The following excerpt comes from InitializeScript() (p. 34):

```lua
GF.ButtonGroup.New(5, 1, "sdButtons")
    :WithContent({
        {"Revert",  btnRevert},
        {"Undo",    btnUndo},
        {"Apply",   btnApply},
        {"Commit",  "ButtonOK"},
        {"Discard", "ButtonCancel"}
    }, 370, 5)
```

Several important conventions are demonstrated.

Vertical layout: Five rows and one column produce a vertical stack.

Custom styling: The additional CSS class `sdButtons` allows the gadget to customize the group's appearance.

Application callbacks: Revert, Undo, and Apply use Lua function references.

Dialog completion: Commit and Discard use Vectric's special button identifiers rather than ordinary programmer-defined callbacks.

Custom button styling

Sectio Divina also defines a CSS rule for its buttons:

```lua
GF.AddStyle([[
.sdButtons #ButtonCancel,
.sdButtons #ButtonOK,
.sdButtons input[type="button"] { width:100px; height:35px; }]])
```

This styles buttons inside the group's container without changing their HTML definitions.

Notice that the special ButtonOK and ButtonCancel identifiers are included explicitly.

Suppressing the default buttons

Because Sectio Divina supplies its own Commit and Discard controls, it calls:

```lua
GF.NoDefaultButtons()
```

This prevents Gadget Factory from appending the normal Execute and Cancel buttons.

The custom button group therefore supplies the dialog's action area.

See Native completion and LuaButton callbacks (p. 289) for the tested distinction.

The Hello-Millie result confirms raw HTML completion without `LuaButton`. It does not directly test the particular ButtonGroup-generated layout shown here, which still needs its own native Aspire exercise. Function actions continue to use the LuaButton callback mechanism.

Preparing and rendering content independently

Like the other group components, ButtonGroup provides methods for separating content preparation from normal dialog construction.

These facilities are useful for visual designers such as Forge and for generating HTML without registering callback handlers.

PrepareContent

```lua
group:PrepareContent(items, left, top)
```

Accepts the same arguments as `WithContent()`.

Return value: The ButtonGroup instance.

The method stores the layout coordinates and prepares an internal representation of each supplied button.

For each item, it determines:

- The displayed label.
- Whether the action is a Lua function.
- The generated or explicitly supplied HTML identifier.
- The optional callback argument.
- The optional tooltip.

It does not register any event handlers or add content to Gadget Factory's dialog.

GetHtml

```lua
local html = group:GetHtml()
```

Return value: A string containing the ButtonGroup's generated HTML.

The method uses the data previously established by `PrepareContent()`.

It produces the outer container, inner row or column containers, and individual buttons.

It does not append the generated markup to Gadget Factory's document or register callbacks.

Example

```lua
local Preview = GF.ButtonGroup.New(1, 2)

Preview:PrepareContent({
    {"Apply", ApplyChanges},
    {"Reset", ResetSettings}
}, 20, 50)

local html = Preview:GetHtml()
```

The resulting HTML contains the two buttons and their generated identifiers.

However, the callbacks are not registered.

Generating ButtonGroup HTML is not equivalent to installing a working ButtonGroup.

For normal gadget construction, use `WithContent()`.

Relationship to WithContent

The current implementation follows this general sequence:

```lua
function ButtonGroup:WithContent(items, left, top)
    self:PrepareContent(items, left, top)

    -- Register function actions
    -- Add generated HTML to Gadget Factory
```

LUA / CONTINUED

```
    return self
end
```

More specifically, function actions are registered through `Core.AddHandler()`, and the generated HTML is appended through `Core.RegisterContent()`.

ButtonGroup does not register itself as a native field-binding object.

This is a useful distinction from the RadioGroup and CheckboxGroup implementations.

How ButtonGroup works internally

ButtonGroup is represented by a Lua object with automatically generated identifiers and a shared metatable.

Every instance stores its grid dimensions, styling class, position, and prepared button definitions.

Automatic identifiers

The first ButtonGroup instance receives an identifier equivalent to:

LUA

```
btng1
```

Its outer container uses:

LUA

```
divbtng1
```

The first button with a Lua function action receives an ID equivalent to:

LUA

```
btn1_btng1
```

These identifiers are generated using Gadget Factory's shared component utilities.

The programmer normally does not need to manage them.

Explicit string actions are an exception because the supplied string becomes the button's ID.

HTML generation

A simple ButtonGroup generates HTML resembling:

HTML

```
<div id="divbtng1" style="left:20px;top:50px;">
    <div class="cssBTNG">

        <div class="cssButtonRow">
```

HTML / CONTINUED

```
            <input type="button" class="LuaButton"
                   id="btn1_btng1" value="Apply">

            <input type="button" class="LuaButton"
                   id="btn2_btng1" value="Reset">
        </div>

    </div>
</div>
```

The outer container establishes the group's position, while the inner containers determine its arrangement.

The group's standard CSS class is `cssBTNG`.

For Lua function actions, the button's `LuaButton` class allows Vectric to generate the appropriate callback event.

The surrounding CSS classes control layout and styling without interfering with that special class.

Callback registration

When `WithContent()` encounters a button whose action is a Lua function, it registers the handler through Gadget Factory's existing mechanism.

The essential implementation is:

LUA

```
VisitButtons(self, function(i)
    local Item = self.content[i]

    if Item and type(Item.action) == "function" then
        Core.AddHandler(Item.id, Item.action, Item.args)
    end
end)
```

No separate global Lua callback function is created for each button.

Gadget Factory's central dispatcher handles the event routing.

This also explains why ButtonGroup does not need its own native `Bind()` method.

Its buttons are event sources, not Vectric-bound data fields.

Styling

The supplied `ButtonGroup.css` stylesheet defines the component's basic layout and button dimensions.

The default button rule is:

```css
.cssBTNG input[type="button"] {
    white-space: nowrap;
    overflow: hidden;
    width: 96px;
    height: 28px;
    margin: 8px;
}
```

The stylesheet also includes rules for the special `ButtonOK` and `ButtonCancel` identifiers, ensuring that they participate in the group's layout rather than retaining their ordinary default-button positioning.

Row and column spacing

The stylesheet defines separate layout rules for rows and columns.

```css
.cssButtonRow + .cssButtonRow {
    margin-top: 8px;
}

.cssButtonColumn + .cssButtonColumn {
    margin-left: 8px;
}
```

Buttons also receive individual margins.

Together, these rules determine the spacing between controls.

Gadget-specific styling

The optional constructor style parameter allows custom classes to be applied to the group's container.

For example:

```lua
GF.ButtonGroup.New(1, 3, "compactActions")
```

The gadget can then provide corresponding CSS:

```lua
GF.AddStyle([[
.cssBTNG.compactActions input[type="button"] {
    width: 88px;
    height: 26px;
    margin: 4px;
}
]])
```

Because the additional class belongs to the container, it does not interfere with the `LuaButton` class on individual callback buttons.

When overriding button dimensions, remember that the stylesheet also contains ID-specific rules for `ButtonOK` and `ButtonCancel`. Those may require corresponding overrides.

The current default styling is not part of ButtonGroup's functional contract. Programmers should use CSS when they need a different appearance rather than depending on fixed dimensions.

ButtonGroup versus selection groups

ButtonGroup shares the same construction pattern as RadioGroup and CheckboxGroup but serves a different purpose.

Feature	ButtonGroup	RadioGroup	CheckboxGroup
Primary purpose	Execute actions	Choose one value	Manage Boolean options
Constructor	`New(rows, cols, style)`	`New(title, onselect)`	`New(title, onselect)`
Item definition	`{label, action, args, title}`	`{label, value}`	`{key, label, checked}`
Native field binding	None	Radio group	Individual checkboxes
Programmatic selection	Not applicable	`Select()`	`SetValues()`
Value retrieval	Not applicable	`Selected()`	`GetValues()`
Event handling	Per-button function action	Optional group callback	Optional group callback

ButtonGroup is fundamentally an action container rather than a selection container.

It has no current selection to retrieve or update.

Each button operates independently, and its behavior is defined by its action.

Important considerations

- **Library loading:** Load ButtonGroup through `GF.LoadComponents()` before creating an instance.
- **Multiple instances:** Every ButtonGroup receives a unique generated group identifier.
- **Grid dimensions:** Supply positive integer row and column counts.
- **Item capacity:** The grid processes positions from `1` through `rows * cols`. Extra items are ignored.
- **Missing items:** Undefined positions do not generate buttons.
- **Item order:** Items are supplied in row-major order, even when the generated HTML uses column containers.
- **Function actions:** Lua function references receive automatically generated button IDs and callback registrations.

- **String actions:** String values are used as literal HTML identifiers. They are not looked up as Lua callback names.
- **Callback arguments:** The optional `args` value is supplied as a single argument, without automatic unpacking.
- **Callback return values:** Gadget Factory's dispatcher handles Vectric's required callback return value.
- **No automatic instance argument:** ButtonGroup does not pass the originating group to its callbacks.
- **Tooltips:** The fourth item value becomes the HTML `title` attribute.
- **Custom styles:** The constructor's `style` argument specifies an additional CSS class, not an inline style string.
- **No native binding:** ButtonGroup does not expose a `Bind()` operation or register native input fields.
- **Single registration:** `WithContent()` should normally be called once per instance. Repeated registration can produce duplicate handlers or markup.
- **No automatic reconfiguration:** ButtonGroup does not currently expose methods for adding, removing, or repositioning individual buttons after construction.
- **Read-only rendering:** `PrepareContent()` and `GetHtml()` generate HTML without registering callbacks.
- **Special identifiers:** Use `ButtonOK` and `ButtonCancel` carefully. They have native dialog significance and must not be duplicated elsewhere in the HTML.
- **Default-button suppression:** Call `GF.NoDefaultButtons()` when replacing the default pair with your own dialog-completion controls.
- **No automatic escaping:** Labels, tooltips, and explicitly supplied identifiers are inserted into HTML without escaping.
- **CSS compatibility:** Final appearance depends on the component stylesheet, Gadget Factory's shared styling, and the embedded Trident engine.

Related functions

- LoadComponents (p. 36) - Load Gadget Factory component libraries.
- Default Buttons (p. 288) - Manage Execute and Cancel behavior.
- Event Handling (p. 320) - Understand Gadget Factory's LuaButton dispatcher.
- RadioGroup (p. 120) - Create mutually exclusive selections.
- CheckboxGroup (p. 137) - Manage independent Boolean selections.
- ToolStrip (p. 173) - Create a dockable collection of action buttons.
- AddStyle (p. 108) - Apply custom CSS styling.
- InitializeScript (p. 34) - Construct and register interface components.

Summary

`GF.ButtonGroup` provides a flexible way to arrange related actions in horizontal, vertical, or multirow grids.

Its public API consists of four methods:

Method	Purpose
`New()`	Create a ButtonGroup and establish its grid dimensions.
`WithContent()`	Define buttons, register function actions, and add HTML to the dialog.
`PrepareContent()`	Prepare button definitions without registering them.
`GetHtml()`	Generate the group's HTML independently.

The component relies on Gadget Factory's centralized event-dispatch system rather than native Vectric field bindings.

Its layout adapts automatically to the specified grid dimensions, while optional CSS classes allow programmers to customize the presentation.

ButtonGroup handles the buttons. The gadget supplies the actions.

REFERENCE 5.5

ToolStrip

`GF.ToolStrip` creates a collection of square action buttons docked to one edge of a Gadget Factory dialog.

Unlike ButtonGroup (p. 156), which arranges buttons in a programmer-defined grid, ToolStrip automatically positions its container against the **left, top, right, or bottom** edge of the interface.

Its orientation is determined by the chosen docking edge:

- Left and Right produce vertical button strips.
- Top and Bottom produce horizontal button strips.

Each button can invoke a Lua callback, optionally receive an argument, and display a tooltip.

ToolStrip is particularly useful for application commands, frequently used operations, and compact interfaces where ordinary action buttons would occupy too much space.

Loading the component

ToolStrip is supplied as a separate library module and must be loaded before use.

```lua
GF.LoadComponents {"ToolStrip"}
```

This exposes the `GF.ToolStrip` constructor and includes the component's `ToolStrip.css` stylesheet.

The module is loaded once per gadget. Multiple ToolStrip instances may then be created independently.

Basic usage

A ToolStrip is constructed using `New()` and populated using `WithContent()`.

```lua
local Tools = GF.ToolStrip.New(36, "left")
    :WithContent({
        {"S", SelectTool, nil, "Select"},
        {"M", MoveTool,   nil, "Move"},
        {"?", ShowHelp,   nil, "Help"}
    })
```

This creates a vertical ToolStrip docked to the left edge of the dialog.

Each button measures 36 × 36 pixels.

The displayed labels are deliberately short because the buttons are square. Their corresponding tooltips provide more descriptive text.

The programmer supplies the actions, while ToolStrip handles positioning, HTML generation, and callback registration.

Construction sequence

The normal construction sequence consists of two operations:

1. `New()` creates the ToolStrip instance and establishes its button size and docking edge.
2. `WithContent()` prepares the buttons, registers any Lua callbacks, generates the HTML, and adds it to Gadget Factory's dialog content.

Both methods return the ToolStrip instance, allowing the fluent construction style shown above.

New

`ToolStrip.New()` creates a new ToolStrip instance.

Syntax

```lua
local strip = GF.ToolStrip.New(size, dock)
```

Parameters

Parameter	Description
`size`	Width and height of each square button, in pixels.
`dock`	Edge of the dialog to which the strip is attached.

Return value: A new ToolStrip instance.

Button size

The `size` parameter determines both dimensions of every button.

```lua
local Tools = GF.ToolStrip.New(32, "left")
```

Each button receives the equivalent of:

```css
width: 32px;
height: 32px;
```

The constructor requires a positive, finite numeric size.

In ordinary use, a positive integer is preferable because it represents a whole number of pixels.

The size does not represent the total thickness of the ToolStrip. Its container also includes CSS padding and borders.

Docking options

The following values are supported:

Value	Position	Orientation
`"left"`	Left edge	Vertical
`"right"`	Right edge	Vertical
`"top"`	Top edge	Horizontal
`"bottom"`	Bottom edge	Horizontal

The values are lowercase and case-sensitive.

For example:

```lua
GF.ToolStrip.New(36, "left")
GF.ToolStrip.New(36, "top")
GF.ToolStrip.New(36, "right")
GF.ToolStrip.New(36, "bottom")
```

An invalid docking value causes an assertion error.

Automatic positioning

Unlike ButtonGroup, ToolStrip does not accept `left` and `top` positioning arguments.

Its location is determined by the docking option and the component's CSS.

A left-docked strip uses:

```css
.cssTSBG.dockLeft {
    left: 0;
    top: 0;
    bottom: 0;
}
```

A top-docked strip uses:

```css
.cssTSBG.dockTop {
    left: 0;
    right: 0;
    top: 0;
}
```

The remaining docking positions use corresponding rules.

ToolStrip's outer container uses `position:absolute`, so these coordinates are interpreted relative to its containing layout.

In normal Gadget Factory construction, this is effectively the dialog's content area.

ToolStrip does not automatically move other controls out of its way. The programmer should reserve sufficient space for the docked strip when positioning the rest of the interface.

WithContent

`WithContent()` defines the buttons and their actions.

Syntax

```lua
strip:WithContent(items)
```

Parameters

Parameter	Description
`items`	Array-style table containing the button definitions.

Return value: The ToolStrip instance.

Item structure

Each item uses the following convention:

```lua
{label, action, args, title}
```

Element	Description
`label`	Text displayed on the button.
`action`	Lua function reference or explicit HTML button identifier.
`args`	Optional argument passed to a Lua callback.
`title`	Optional HTML tooltip.

The label and action are required for normal use.

For example:

```lua
{"R", RefreshPreview}
```

Creates a button displaying `R` that invokes `RefreshPreview()`.

Supplying a tooltip

```lua
{"R", RefreshPreview, nil, "Refresh the preview"}
```

The fourth element becomes the button's HTML `title` attribute.

The `nil` placeholder indicates that no callback argument is required.

ToolStrip explicitly unpacks all four item positions:

```lua
local label, action, args, title = table.unpack(item, 1, 4)
```

Consequently, the missing third argument does not prevent the fourth argument from being retrieved.

This is the same convention used by ButtonGroup (p. 156).

Button count

The number of buttons is determined entirely by the supplied item collection.

ToolStrip does not require row or column counts, and its constructor does not impose a fixed button capacity.

For example:

```lua
GF.ToolStrip.New(32, "top")
    :WithContent({
        {"A", ActionA},
        {"B", ActionB},
        {"C", ActionC},
        {"D", ActionD}
    })
```

Creates four horizontally arranged buttons.

Additional items can be added without changing the constructor.

The practical limit is the available space in the containing interface.

Button actions

ToolStrip distinguishes two action types: Lua functions and explicit HTML identifiers.

Lua function actions

When the action is a Lua function, ToolStrip generates a unique button ID, assigns the special `LuaButton` class, and registers the function through Gadget Factory's event dispatcher.

For example:

```lua
{"Refresh", RefreshPreview}
```

Creates HTML resembling:

```html
<input type="button" class="LuaButton"
       id="tsi1_tsbg1"
       value="Refresh"
       style="width:36px;height:36px;">
```

The generated identifier depends on the ToolStrip instance and the item's position.

The callback is registered through Gadget Factory's AddHandler (p. 320) mechanism.

When the button is activated, Vectric invokes Gadget Factory's dispatcher, which executes the supplied function.

The programmer does not need to define an individual global Lua callback.

Passing arguments

Several buttons may invoke the same function with different arguments.

```lua
local ChangeMode = function(mode)
    CurrentMode = mode
end

GF.ToolStrip.New(36, "left")
    :WithContent({
        {"S", ChangeMode, "Select", "Selection mode"},
        {"M", ChangeMode, "Move",   "Movement mode"},
        {"D", ChangeMode, "Draw",   "Drawing mode"}
    })
```

Selecting the `M` button effectively invokes:

```lua
ChangeMode("Move")
```

The optional `args` value is passed as **one argument**.

It is not automatically unpacked.

Passing multiple values

Several values may be packaged into a Lua table.

```lua
local ApplySettings = function(options)
    SetMode(options.mode)
    SetPrecision(options.precision)
end

GF.ToolStrip.New(36, "top")
    :WithContent({
        {"F", ApplySettings, {
            mode = "Fine",
            precision = 0.001
        }, "Fine mode"},

        {"C", ApplySettings, {
            mode = "Coarse",
            precision = 0.01
        }, "Coarse mode"}
    })
```

Each callback receives its corresponding table.

The action function can then retrieve the supplied values using ordinary Lua table syntax.

Explicit identifier actions

When `action` is not a function, ToolStrip uses its value directly as the HTML button ID.

For example:

```lua
{"OK", "ButtonOK"}
{"X",  "ButtonCancel"}
```

These identifiers have special significance in Vectric's dialog-completion mechanism.

However, this action type behaves differently from a Lua function action.

The current ToolStrip implementation:

- Uses the supplied string directly as the HTML ID.
- Does not generate an automatic item ID.
- Does not register a Gadget Factory event handler.
- Does not assign the `LuaButton` class.

The intended use is for buttons whose identifiers carry their own special meaning.

An arbitrary string is not interpreted as the name of a Lua callback function.

For example:

```lua
{"R", "RefreshPreview"}
```

does not register or invoke a Lua function named `RefreshPreview`.

Supply the function reference instead:

```lua
{"R", RefreshPreview}
```

In the Hello-Millie Aspire test (p. 289), raw HTML buttons using `ButtonOK` and `ButtonCancel` completed the dialog without `LuaButton`. The absence of that class in ToolStrip's string-action markup is therefore not, by itself, a reason to expect completion to fail.

That test did not exercise a particular ToolStrip-generated layout. Verify that layout separately in the target Aspire environment. Ordinary function actions continue to use the LuaButton callback mechanism.

Docking and layout

ToolStrip generates one continuous row or column of buttons.

Its orientation depends entirely on its docking edge.

Vertical ToolStrips

Left and Right docking use vertical button arrangements.

```lua
GF.ToolStrip.New(36, "left")
```

The buttons are stacked vertically.

The component's CSS uses:

```css
.cssTSBG.dockLeft input[type="button"],
.cssTSBG.dockRight input[type="button"] {
    display: block;
}
```

Adjacent buttons receive a small vertical margin.

The strip's outer container extends between the top and bottom edges of its containing layout.

Horizontal ToolStrips

Top and Bottom docking use horizontal button arrangements.

```lua
GF.ToolStrip.New(36, "top")
```

The buttons are displayed horizontally.

The component's stylesheet uses `inline-block` for the buttons and applies horizontal margins between adjacent items.

The outer container extends from the left to the right edge of its containing layout.

Avoiding HTML whitespace

The component deliberately concatenates its button markup without inserting text nodes between adjacent buttons.

Its rendering function concludes with:

```lua
return table.concat(html)
```

The distinction matters because HTML whitespace can introduce unwanted gaps between inline-block elements.

By eliminating those gaps, ToolStrip leaves spacing entirely under CSS control.

This is especially useful for horizontal toolbars where consistent button spacing is important.

Multiple ToolStrips

Multiple ToolStrip instances may be created in the same gadget.

For example:

```lua
GF.ToolStrip.New(32, "left")
    :WithContent({
        {"S", SelectTool},
        {"M", MoveTool}
    })

GF.ToolStrip.New(32, "top")
    :WithContent({
        {"R", RefreshPreview},
        {"?", ShowHelp}
    })
```

However, **ToolStrip does not provide a docking manager**.

Two strips docked to the same edge may overlap, and strips docked to adjoining edges may compete for space at their corners.

The programmer remains responsible for coordinating layouts when several strips are present.

For example, a top-docked strip and a left-docked strip may need additional CSS adjustments to prevent their containers from overlapping.

Similarly, a bottom-docked ToolStrip may overlap StatusBar (p. 298) unless the surrounding layout accounts for both.

Example: A compact command strip

The following example demonstrates a left-docked ToolStrip used to control an application's operating mode.

```lua
local CurrentMode = "Select"

local ChangeMode = function(mode)
    CurrentMode = mode
    GF.Dialog:UpdateLabelField("labMode", "Mode: " .. mode)
end

local ShowHelp = function()
    GF.Dialog:UpdateLabelField(
        "labMode",
        "Choose a tool from the strip"
    )
end

local function InitializeScript(job)

    GF.LoadComponents {"ToolStrip"}

    GF.AddDiv('style="margin-left:56px;"', {
        {GF.AddLabelSpan, "labMode", "Mode: " .. CurrentMode}
    })

    GF.ToolStrip.New(36, "left")
        :WithContent({
            {"S", ChangeMode, "Select", "Selection mode"},
            {"M", ChangeMode, "Move",   "Movement mode"},
            {"?", ShowHelp,   nil,      "Show help"}
        })
end
```

When the dialog opens, the ToolStrip appears along its left edge.

The AddDiv (p. 80) wrapper keeps the label out of the strip's area. Its 56-pixel left margin allows for the approximately 46-pixel strip thickness with the supplied stylesheet, plus a small gap. Adjust this margin if you change the button size or styling.

Selecting a mode changes the application variable and updates the displayed label.

The Help button displays a short instruction without requiring a modal message box.

This example is illustrative; the actual application would normally perform additional work when its operating mode changes.

Tooltips and compact labels

Square buttons often lack sufficient space for long descriptive labels.

ToolStrip's `title` parameter provides a convenient alternative.

For example:

```lua
{"R", RefreshPreview, nil, "Recalculate the preview"}
```

The button displays `R`, while hovering reveals the complete description.

Unicode symbols and emoji may also be used as labels, subject to the rendering capabilities of the embedded browser and installed fonts.

ToolStrip preserves their original UTF-8 bytes when generating HTML.

Preparing and rendering content independently

Like the other group components, ToolStrip exposes methods for separating content preparation and HTML generation from normal Gadget Factory registration.

These methods are particularly useful for interface designers such as Forge.

PrepareContent

```lua
strip:PrepareContent(items)
```

Accepts the same item table as `WithContent()`.

Return value: The ToolStrip instance.

The method prepares each button's label, action, ID, optional argument, and tooltip.

Function actions receive automatically generated identifiers based on the ToolStrip instance and item index.

String actions retain their supplied identifiers.

The prepared definitions are stored for subsequent rendering.

`PrepareContent()` does not register event handlers or add HTML to the gadget's document.

GetHtml

```lua
local html = strip:GetHtml()
```

Return value: A string containing the complete ToolStrip HTML.

This includes the outer docked container, the inner button container, and all prepared buttons.

The method does not append content to Gadget Factory or register callbacks.

`PrepareContent()` or `WithContent()` must have prepared the item definitions before `GetHtml()` is used.

Example

```lua
local Preview = GF.ToolStrip.New(32, "top")

Preview:PrepareContent({
    {"A", ActionA},
    {"B", ActionB}
})

local html = Preview:GetHtml()
```

The returned string contains the HTML for a top-docked strip with two buttons.

However, **the Lua callbacks have not been registered**.

This is appropriate when rendering a preview or inspecting the generated markup.

For a normal working gadget, use `WithContent()` instead.

Relationship to WithContent

The current implementation follows this sequence:

```lua
function ToolStrip:WithContent(items)
    self:PrepareContent(items)

    for _, item in ipairs(PreparedItems[self]) do
        if type(item.action) == "function" then
            Core.AddHandler(item.id, item.action, item.args)
        end
    end

    Core.RegisterContent(nil, self:GetHtml())

    return self
end
```

This illustrates the separation between preparing definitions, registering behavior, and generating HTML.

Notice the use of:

```lua
Core.RegisterContent(nil, self:GetHtml())
```

The first argument is `nil` because ToolStrip does not need a native Vectric field binding.

Its function buttons communicate through Gadget Factory's existing LuaButton event dispatcher.

This is another distinction from RadioGroup and CheckboxGroup, which register native selection fields.

Prepared content and runtime changes

The separation between `PrepareContent()` and `GetHtml()` makes it possible to generate alternative HTML representations.

However, the current public API does not provide a dedicated method for changing the buttons of an already displayed ToolStrip.

Calling `PrepareContent()` again does not automatically replace the visible HTML or register new handlers.

Therefore, these methods should not be confused with a supported runtime reconfiguration system.

Gadget Factory's Dynamic HTML (p. 364) mechanisms could support specialized solutions, but programmers must manage the associated identities and callbacks correctly.

How ToolStrip works internally

ToolStrip uses a Lua object with automatically generated identifiers and a shared metatable.

Its implementation is compact because most of the difficult work - HTML collection, styling, and callback dispatching - is delegated to existing Gadget Factory facilities.

Automatic identifiers

The first ToolStrip instance receives an identifier equivalent to:

```lua
tsbg1
```

Its outer HTML container uses:

```lua
divtsbg1
```

Its inner item container uses:

```lua
items_tsbg1
```

The first button whose action is a Lua function receives an identifier equivalent to:

```lua
tsi1_tsbg1
```

The `tsbg` prefix identifies the ToolStrip group, while `tsi` identifies individual ToolStrip items.

These IDs are generated using Gadget Factory's shared component-identifier facility.

The programmer normally works with the ToolStrip instance and its action functions rather than constructing these identifiers manually.

Generated HTML

A simplified left-docked ToolStrip generates markup resembling:

```HTML
<div id="divtsbg1" class="cssTSBG dockLeft">
    <div id="items_tsbg1"
         class="cssToolStripItems"
         style="width:36px;">

        <input type="button" class="LuaButton"
               id="tsi1_tsbg1"
               value="S"
               title="Selection mode"
               style="width:36px;height:36px;">

    </div>
</div>
```

The actual generated markup places the button elements consecutively without additional whitespace between them.

The outer container establishes the docking edge.

The inner container establishes the appropriate cross-axis dimension.

For vertical strips, its width equals the configured button size.

For horizontal strips, its height equals the configured button size.

CSS padding and borders determine the additional thickness of the outer shell.

HTML attribute escaping

ToolStrip uses the framework's shared `Core.EscapeHtml()` utility to escape generated button IDs, labels, and tooltips before inserting them into HTML attributes.

The escaping implementation belongs to the framework; ToolStrip does not maintain a separate local helper. Gadget code uses the public `GF.EscapeHtml()` utility, as demonstrated in A Preview Workbench (p. 400).

This prevents characters such as quotation marks and angle brackets from accidentally breaking the surrounding HTML attributes.

For example:

```LUA
{"?", ShowHelp, nil, 'Help with "Advanced" options'}
```

The quotation marks inside the tooltip are converted to HTML entities.

The resulting attribute is equivalent to:

```LUA
title="Help with &quot;Advanced&quot; options"
```

The browser displays the original quotation marks when presenting the tooltip.

ToolStrip performs attribute escaping, not general HTML sanitization.

Labels and tooltips are treated as text rather than HTML markup. Programmers should supply strings suitable for the respective fields.

Unicode characters are preserved rather than converted into replacement characters or escape sequences.

Prepared item storage

ToolStrip stores prepared button definitions in a private table indexed by component instance.

The implementation uses a weak-key table:

```lua
local PreparedItems = setmetatable({}, { __mode = "k" })
```

In Lua, weak table keys do not prevent their referenced objects from being garbage-collected.

This allows the module to associate prepared definitions with their ToolStrip instances without unnecessarily keeping discarded instances alive.

The detail is normally invisible to the gadget programmer, but it illustrates another useful application of Lua's table and metatable facilities.

Styling

The supplied `ToolStrip.css` stylesheet controls docking, button dimensions, spacing, and visual states.

Outer container

The main ToolStrip container uses:

```css
.cssTSBG {
    position: absolute;
    box-sizing: border-box;
    padding: 4px;
    border: 1px solid #b8b8b8;
    background-color: #e8e8e8;
}
```

This creates a restrained gray background and a subtle border separating the strip from the surrounding interface.

With the current stylesheet, a strip containing 36-pixel buttons has a cross-axis thickness of approximately 46 pixels: 36 pixels for the button, 8 pixels of padding, and 2 pixels of border.

The calculation assumes that no gadget-specific CSS overrides these dimensions.

Button appearance

The button styling uses a light-gray gradient, a one-pixel border, and subtle rounded corners.

The current stylesheet also defines hover, pressed, and keyboard-focus appearances.

These states are provided through CSS rather than Lua callbacks.

For example:

```
CSS
.cssTSBG .cssToolStripItems input[type="button"]:hover {
    background-color: #efefef;
    background-image: linear-gradient(
        to bottom,
        #ffffff 0%,
        #e5e5e5 100%
    );
    border-color: #888888;
}
```

The pressed state uses a slightly darker appearance, while the focus state uses a dotted outline.

ToolStrip does not maintain a persistent selected-button state.

The hover and pressed effects represent ordinary button interaction, not application mode selection.

Button spacing

The stylesheet provides three pixels of spacing between adjacent buttons.

For vertical strips:

```
CSS
.cssTSBG.dockLeft input[type="button"] +
input[type="button"],
.cssTSBG.dockRight input[type="button"] +
input[type="button"] {
    margin-top: 3px;
}
```

Horizontal strips use the corresponding `margin-left` rule.

These spacing values are defined in CSS and are not constructor parameters.

Gadget-specific customization

The programmer can override ToolStrip's appearance through AddStyle (p. 108).

For example:

```lua
GF.AddStyle([[
    .cssTSBG {
        background-color: #f2f2f2;
        border-color: #c5c5c5;
    }

    .cssTSBG .cssToolStripItems input[type="button"] {
        color: #214070;
    }
]])
```

Because the component's CSS uses relatively specific selectors, overrides may need equivalent or greater specificity.

The stylesheet also contains explicit rules for special button IDs such as `ButtonOK` and `ButtonCancel`, preventing Gadget Factory's general button styling from interfering with the ToolStrip layout.

As with the other components, its presentation can be customized without modifying the library's Lua implementation.

ToolStrip versus ButtonGroup

Both components create collections of action buttons, but their intended layouts differ.

Feature	ToolStrip	ButtonGroup
Layout	Docked row or column	Programmer-defined grid
Positioning	Edge docking	Explicit Left and Top
Button dimensions	Square, set by constructor	CSS-controlled
Item definition	`{label, action, args, title}`	`{label, action, args, title}`
Callback registration	Gadget Factory dispatcher	Gadget Factory dispatcher
Native field binding	None	None
Custom CSS class argument	Not provided	Supported
Number of items	Determined by item table	Limited by grid capacity

The identical item structure makes it relatively easy to use the same action definitions in either component.

For example, a collection of application commands could be displayed as a ButtonGroup in one gadget and as a docked ToolStrip in another.

The choice depends primarily on the desired layout and available interface space.

Use ButtonGroup for an independent group of actions. Use ToolStrip for commands that belong along an edge of the interface.

Important considerations

- **Library loading:** Load ToolStrip before creating an instance.
- **Multiple instances:** Each instance receives independently generated identifiers.
- **Button size:** The constructor accepts a positive, finite numeric size in pixels.
- **Docking values:** Use exactly `"left"`, `"right"`, `"top"`, or `"bottom"`.
- **No manual coordinates:** The public API does not accept Left or Top arguments.
- **Layout space:** ToolStrip does not automatically reserve space for adjacent content.
- **Multiple docked strips:** Strips may overlap when assigned competing edges or positions.
- **Square buttons:** Width and height are always generated from the same size argument.
- **Unlimited item definitions:** There is no fixed item count in the constructor, although available display space is finite.
- **Function actions:** Lua functions receive generated IDs, the `LuaButton` class, and registered handlers.
- **String actions:** Strings become literal button IDs and do not register ordinary Lua callbacks.
- **Required item types:** Labels and tooltips should be strings. Actions should be functions or valid HTML ID strings.
- **Callback arguments:** The optional `args` value is passed as one argument without automatic unpacking.
- **No automatic instance argument:** ToolStrip does not supply its own instance to callbacks.
- **Tooltips:** The fourth item entry becomes the HTML `title` attribute.
- **Attribute escaping:** Labels, IDs, and tooltips are escaped for safe insertion into HTML attributes.
- **No persistent selection:** Buttons do not maintain selected or toggled states.
- **Single registration:** `WithContent()` should normally be called once per instance. Repeated calls can produce duplicate handlers and markup.
- **Read-only rendering:** `PrepareContent()` and `GetHtml()` generate HTML without installing callback handlers.
- **No public redraw method:** Runtime replacement of a ToolStrip's contents is not automatically managed by this version of the component.
- **Native completion buttons:** Raw HTML completion without `LuaButton` is confirmed in the Hello-Millie test (p. 289). A particular ToolStrip-generated layout still needs its own native exercise.
- **Browser compatibility:** Appearance and interaction are subject to the capabilities of Vectric's embedded Trident engine.

Related functions

- LoadComponents (p. 36) - Load Gadget Factory component libraries.
- ButtonGroup (p. 156) - Create freely positioned grids of action buttons.
- Event Handling (p. 320) - Understand Gadget Factory's LuaButton dispatcher.
- Default Buttons (p. 288) - Manage standard dialog-completion controls.
- AddStyle (p. 108) - Customize the component's appearance.
- Dynamic HTML (p. 364) - Understand runtime HTML replacement and control identity.
- InitializeScript (p. 34) - Construct and register interface components.

Summary

`GF.ToolStrip` creates docked collections of square action buttons for compact, command-oriented interfaces.

Its public methods are:

Method	Purpose
`New()`	Create a ToolStrip with a specified button size and docking edge.
`WithContent()`	Define buttons, register actions, and add the strip to the dialog.
`PrepareContent()`	Prepare button definitions without registering them.
`GetHtml()`	Generate the ToolStrip's HTML independently.

ToolStrip uses Gadget Factory's event-dispatching mechanism for Lua function actions and CSS for docking, spacing, and visual feedback.

It does not require native Vectric field bindings or expose selection-management methods.

ButtonGroup organizes actions within the interface. ToolStrip organizes actions around its edges.

REFERENCE 5.6

Combobox

`GF.Combobox` creates a dropdown selection control whose items and selected value can be managed programmatically.

It provides functionality similar to Vectric's native dropdown control while adding capabilities for modifying the available choices during dialog execution.

Each item consists of a descriptive label and an associated Lua value. The programmer works with these values rather than the internal identifiers used by the HTML control.

Combobox supports multiple independent instances, optional selection callbacks, explicit empty selections, and methods for adding, removing, replacing, or rebuilding its item collection.

Unlike RadioGroup (p. 120), which displays all its choices simultaneously, Combobox presents them through a compact HTML dropdown.

Loading the component

Combobox is supplied as a separate library module and must be loaded before use.

```lua
GF.LoadComponents {"Combobox"}
```

This exposes the `GF.Combobox` constructor and includes the companion `Combobox.css` stylesheet.

The module is loaded once per gadget. Multiple Combobox instances may then be created.

Basic usage

A Combobox is created using `New()` and populated using `WithContent()`.

```lua
local Materials = GF.Combobox.New("Material")
    :WithContent({
        {"MDF",       "mdf"},
        {"Plywood",   "plywood"},
        {"Hardwood", "hardwood"}
    }, 30, 60, 240)
    :Select(1)
```

This creates a dropdown titled **Material**, positioned 30 pixels from the left and 60 pixels from the top, with a width of 240 pixels.

The first item, MDF, is initially selected.

Each item uses the familiar convention:

```lua
{label, value}
```

The displayed label identifies the choice to the user, while the associated value is returned to the gadget.

Construction sequence

The normal sequence consists of three operations:

1. `New()` creates the Combobox instance and establishes an optional callback.
2. `WithContent()` prepares the available items, generates the HTML, and registers the component.
3. `Select()` establishes an initial selection.

All three methods support fluent construction.

Unlike some earlier group components, Combobox does not automatically select the first item when no selection is specified. An unselected Combobox reports index `0`.

New

`Combobox.New()` creates a new Combobox instance.

Syntax

```lua
local combo = GF.Combobox.New(title, onselect)
```

Parameters

Parameter	Description
`title`	Optional heading displayed above the dropdown.
`onselect`	Optional Lua function invoked when the user changes the selection.

Return value: A new Combobox instance.

Without a callback

```lua
local Materials = GF.Combobox.New("Material")
```

The dropdown operates normally, and the selected value can be retrieved when needed.

This is appropriate when the selection will be examined only during final processing.

With a callback

```lua
local MaterialChanged = function(value, combo)
    SelectedMaterial = value
end

local Materials = GF.Combobox.New("Material", MaterialChanged)
```

The callback receives two arguments:

Argument	Description
`value`	Lua value associated with the newly selected item.
`combo`	Combobox instance that generated the event.

The second argument allows one callback to serve multiple Combobox instances.

The constructor requires `onselect` to be a function or `nil`. Other argument types cause an assertion error.

WithContent

`WithContent()` establishes the available choices, layout, and native binding and event registrations.

Syntax

```lua
combo:WithContent(items, left, top, width)
```

Parameters

Parameter	Description
`items`	Array-style table containing `{label, value}` pairs.
`left`	Optional horizontal position in pixels. Defaults to `0`.
`top`	Optional vertical position in pixels. Defaults to `0`.
`width`	Optional width in pixels. Defaults to `200`.

Return value: The Combobox instance.

Item definitions

Each item contains a displayed label and its associated Lua value.

```lua
local Direction = GF.Combobox.New("Cut Direction")
    :WithContent({
        {"Climb",        "CLIMB"},
        {"Conventional", "CONVENTIONAL"}
    }, 20, 50, 250)
```

Selecting Conventional produces the value `"CONVENTIONAL"`.

The values need not be strings.

```lua
local Depth = GF.Combobox.New("Pass Depth")
    :WithContent({
        {"Shallow", 0.02},
        {"Medium",  0.03},
        {"Deep",    0.05}
    }, 20, 50, 250)
```

The selected numeric value can be used directly by the gadget.

Value types

Combobox stores associated values in Lua rather than embedding them directly into HTML.

Consequently, values may include numbers, strings, Booleans, and other Lua values.

The implementation even permits `nil` values, although they deserve special consideration when determining whether an item is selected.

Labels must be strings. Invalid item structures or non-string labels cause an assertion error.

Geometry validation

The `left`, `top`, and `width` arguments must be finite numbers.

Width must also be greater than zero.

The component does not require integer coordinates, although whole-pixel values are conventional.

Registration

`WithContent()`:

1. Prepares the item definitions and internal identifiers.
2. Registers handlers for selection changes and document readiness.
3. Generates the component's HTML.
4. Registers the component with Gadget Factory for native binding.

It may be called only once per instance. A second call raises an assertion error.

To change the available choices after construction, use the item-management methods described below.

Select

`Select()` establishes or changes the selected item by its position in the collection.

Syntax

```lua
combo:Select(index)
```

Parameters

Parameter	Description
`index`	One-based item index, or `0` for no selection.

Return value: The Combobox instance.

Example

```lua
Materials:Select(2)
```

Selects the second item.

The argument is an item index, not its associated value.

For example:

```lua
local Depth = GF.Combobox.New("Pass Depth")
    :WithContent({
        {"Shallow", 0.02},
        {"Medium",  0.03},
        {"Deep",    0.05}
    }, 20, 50, 250)

Depth:Select(2)
```

The selected value becomes `0.03`.

Passing `0.03` to `Select()` would be invalid because the function expects an integer index.

Clearing the selection

Unlike RadioGroup (p. 120), Combobox explicitly supports index zero.

```lua
Materials:Select(0)
```

This clears the selection.

Calling `Select()` without an argument has the same effect.

```lua
Materials:Select()
```

Neither operation removes the available choices.

If the index is outside the valid range, Combobox raises an assertion error rather than silently ignoring the request.

Programmatic selection and callbacks

`Select()` changes the component's selected state and updates the visible dropdown when the dialog is ready.

It does not invoke the programmer's `onselect` callback.

If a programmatic selection must also trigger application logic, call that logic separately.

AllowEmptySelection

`AllowEmptySelection()` adds a visible blank choice to the dropdown.

Syntax

```lua
combo:AllowEmptySelection()
```

Parameters: None.

Return value: The Combobox instance.

Example

```lua
local Destination = GF.Combobox.New("Destination")
    :AllowEmptySelection()
    :WithContent({
        {"Current Layer", "current"},
        {"New Layer",     "new"}
    }, 20, 50, 250)
    :Select(0)
```

The dropdown begins with an empty choice, followed by the two defined items.

The blank choice represents no selection.

Empty selection versus no selected item

This is an important distinction.

`Select(0)` is valid even when `AllowEmptySelection()` has not been called.

Without the optional blank choice, Combobox attempts to display no selected item.

With `AllowEmptySelection()`, the user can deliberately return to an empty selection through the dropdown itself.

The blank choice is not included in the programmer's item collection and does not affect the one-based indices of the defined items.

Enabling the blank option later

`AllowEmptySelection()` may also be called after the component has been installed.

If the dialog is already ready, Combobox refreshes the dropdown to include the blank option.

The method does not provide a corresponding operation for removing this option once enabled.

Selected

`Selected()` retrieves the current selection.

Syntax

```lua
local value, index = combo:Selected()
```

Return values

Return	Description
`value`	Lua value associated with the selected item, or `nil` when none is selected.
`index`	One-based selected index, or `0` when no item is selected.

Example

```lua
local value, index = Materials:Selected()
```

If Plywood is selected:

```lua
-- value = "plywood"
-- index = 2
```

Retrieving only the value

```lua
local material = Materials:Selected()
```

Lua retains only the first return value.

Retrieving only the index

```lua
local _, index = Materials:Selected()
```

This follows the same convention demonstrated by RadioGroup (p. 120).

No selection

When no item is selected:

```lua
local value, index = Materials:Selected()

-- value = nil
-- index = 0
```

A subtle distinction arises when a defined item itself has a `nil` value.

In that case, `value` may also be `nil`, but `index` will identify the selected item.

Use the index when it is necessary to distinguish an unselected dropdown from a selected item whose value is `nil`.

Native state

While the dialog is running and the component is ready, `Selected()` reads the current selection through its Vectric-backed state field.

This ensures that the returned value reflects the user's current choice.

SelectedValue

`SelectedValue()` retrieves the associated Lua value using the component's currently stored selection index.

Syntax

```lua
local value = combo:SelectedValue()
```

Return value: The associated Lua value, or `nil` if no valid item is selected.

Example

```lua
local material = Materials:SelectedValue()
```

Unlike `Selected()`, this method does not first refresh the selection from Vectric.

It simply uses the component's stored index to retrieve the corresponding value.

Consequently, `Selected()` **is generally preferable when the programmer needs to read the current native dialog state**.

`SelectedValue()` is useful when the selection has already been synchronized or established programmatically.

Managing items dynamically

One of Combobox's most significant advantages over Vectric's native dropdown facility is that its item collection can be changed while the dialog is running.

The component provides six methods:

Method	Purpose
`SetItems()`	Replace the entire item collection.
`AddItem()`	Insert or append an item.
`RemoveItem()`	Remove an item by index.
`ReplaceItem()`	Replace an existing item.
`Clear()`	Remove every item.
`GetItems()`	Retrieve a copy of the current item definitions.

A seventh method, `Count()`, reports the number of defined items.

These methods operate on the component's internal item collection and refresh the displayed dropdown when the dialog is ready.

SetItems

`SetItems()` replaces the available choices with a new collection.

```lua
combo:SetItems(items)
```

Return value: The Combobox instance.

For example:

```lua
Materials:SetItems({
    {"Oak",    "oak"},
    {"Maple",  "maple"},
    {"Walnut", "walnut"}
})
```

The dropdown now contains the three hardwood choices.

The previous item collection is replaced.

Preserving the selection

`SetItems()` attempts to preserve the identity of existing items.

It first matches new entries against old entries using both label and value.

If an exact match is unavailable, it attempts to match entries using their non-`nil` Lua values.

This means that changing a label does not necessarily discard the selection.

For example:

```lua
Materials:SetItems({
    {"Plywood", "plywood"},
    {"MDF",     "mdf"}
})
```

If Plywood was previously selected, its selection can survive even though its position has changed.

Likewise, changing its displayed label to `"Cabinet Plywood"` while retaining the value `"plywood"` preserves its identity.

When values occur more than once, matching proceeds in occurrence order so individual items remain distinguishable.

If the previously selected item cannot be matched, the selection is cleared.

Nil-valued items

Because `nil` is not a distinct value suitable for identifying an item across changes, entries with `nil` values retain their identities only through exact label-and-value matches.

Changing the label of a selected nil-valued item can therefore clear its selection.

For selections that must survive label changes, use stable non-`nil` values.

AddItem

`AddItem()` inserts a new item at a specified position or appends it to the collection.

```lua
combo:AddItem(pair, index)
```

Parameters

Parameter	Description
`pair`	New `{label, value}` item.
`index`	Optional one-based insertion position. Defaults to the end.

Return value: The Combobox instance.

Append an item:

```lua
Materials:AddItem({"Birch", "birch"})
```

Insert one at position two:

```lua
Materials:AddItem({"Cherry", "cherry"}, 2)
```

If an item is already selected, inserting before it adjusts its stored index so that the same logical item remains selected.

The permitted insertion positions range from `1` through `Count() + 1`.

RemoveItem

`RemoveItem()` removes an item by its one-based index.

```lua
combo:RemoveItem(index)
```

Return value: The Combobox instance.

Example:

```lua
Materials:RemoveItem(2)
```

When an item before the current selection is removed, the selected index is adjusted to preserve the same selected item.

If the selected item itself is removed, the selection is cleared.

Invalid indices cause an assertion error.

ReplaceItem

`ReplaceItem()` changes an existing item's label and associated value.

```lua
combo:ReplaceItem(index, pair)
```

Return value: The Combobox instance.

Example:

```lua
Materials:ReplaceItem(2, {"Hard Maple", "maple"})
```

The replacement behavior depends on the item's identity.

If the associated non-`nil` value remains unchanged, changing only its label preserves the selection.

If the associated value changes, the replacement is treated as a different item, and a selection on the original item is cleared.

For items whose values are `nil`, both the label and value must remain unchanged to preserve identity.

This prevents an item with a new meaning from silently inheriting the previous item's selection.

Clear

`Clear()` removes every item from the dropdown.

```lua
combo:Clear()
```

Parameters: None.

Return value: The Combobox instance.

The method is equivalent to:

```lua
combo:SetItems({})
```

The selection becomes zero.

If an empty option was previously enabled through `AllowEmptySelection()`, that option remains part of the rendered dropdown.

GetItems

`GetItems()` returns the current item definitions.

```lua
local items = combo:GetItems()
```

Return value: A new array containing `{label, value}` pairs.

For example:

```lua
local items = Materials:GetItems()

for _, item in ipairs(items) do
    local label = item[1]
    local value = item[2]
end
```

The returned array and its item-pair tables are newly constructed, so inserting, removing, or replacing those pairs does not directly modify the Combobox.

However, values that are themselves tables or other reference types are not deep-copied.

To apply a modified collection, pass it to `SetItems()`.

Count

`Count()` reports the number of items currently defined.

```lua
local count = Materials:Count()
```

Return value: Integer representing the number of items.

The optional blank selection is not counted because it is not a programmer-defined item.

`Count()` may be called before or after native dialog binding because it reads the component's Lua item collection.

Selection callbacks

The optional `onselect` function is invoked when the user changes the selection to a different item.

Consider:

```lua
local MaterialChanged = function(value, combo)
    SelectedMaterial = value
end
```

Construct the Combobox using this function:

```lua
local Materials = GF.Combobox.New("Material", MaterialChanged)
    :WithContent({
        {"MDF",       "mdf"},
        {"Plywood",   "plywood"},
        {"Hardwood", "hardwood"}
    }, 30, 60, 240)
```

When the user selects Hardwood, the callback effectively receives:

```lua
MaterialChanged("hardwood", Materials)
```

The associated Lua value is passed directly to the programmer.

Shared callbacks

Several Combobox instances can use the same callback:

```lua
local SelectionChanged = function(value, combo)
    if combo == MaterialBox then
        SelectedMaterial = value
    elseif combo == LayerBox then
        SelectedLayer = value
    end
end
```

The second argument identifies the originating component.

Callback behavior

Combobox distinguishes user-generated selection changes from programmatic updates.

Methods such as `Select()`, `SetItems()`, and `ReplaceItem()` do not explicitly invoke the selection callback.

The callback is intended to execute only when a user interaction produces a different selection.

Internally, Combobox compares the previous and current item identifiers rather than merely comparing associated Lua values.

Consequently, selecting a different item with the same associated value still represents a selection change.

This matters when duplicate values are deliberately used for distinct choices.

Example: A dynamically populated layer selector

The following illustrative example shows how Combobox might be used when one selection determines the available choices in another.

```lua
local LayerBox

local function SetDestinationType(kind)
    if kind == "Existing" then
        LayerBox:SetItems({
            {"Cutting",  "Cutting"},
            {"Engraving", "Engraving"},
            {"Pockets",  "Pockets"}
        })
    else
        LayerBox:SetItems({
            {"New Layer", "NewLayer"}
```

LUA / CONTINUED

```
        })
    end
end

local function InitializeScript(job)
    GF.LoadComponents {"Combobox"}

    LayerBox = GF.Combobox.New("Destination Layer")
        :AllowEmptySelection()
        :WithContent({}, 250, 60, 220)

    GF.Combobox.New("Destination Type", SetDestinationType)
        :WithContent({
            {"Existing Layer", "Existing"},
            {"New Layer",      "New"}
        }, 20, 60, 200)
end
```

Once the dialog is running, selecting Existing Layer populates the destination dropdown with the available examples.

Selecting New Layer replaces the items with a single alternative.

The item lists are illustrative; a working gadget could construct them from the current Vectric job's layer information.

The important capability is that `LayerBox` is modified **without recreating the dialog or registering a new native dropdown field**.

This is the principal reason Combobox maintains its own item collection rather than relying entirely on Vectric's native dropdown population methods.

Preparing and rendering content independently

Like the other Gadget Factory components, Combobox separates content preparation from normal registration.

PrepareContent

LUA

```
combo:PrepareContent(items, left, top, width)
```

Accepts the same arguments as `WithContent()`.

Return value: The Combobox instance.

The method validates the geometry, prepares the item collection, and stores the layout properties.

It does not register event handlers or add the component to Gadget Factory's document.

It uses `SetItems()` internally, so an existing prepared selection may be preserved through the same item-matching rules described earlier.

On an already live Combobox, preparation can also change the visible items. Once the instance is ready, `SetItems()` refreshes its content in the browser. The layout properties are stored afterward; `PrepareContent()` is not a supported way to move or resize a live control.

Use a separate, uninstalled instance when preparing HTML for a preview.

GetHtml

```lua
local html = combo:GetHtml()
```

Return value: String containing the complete Combobox HTML.

The returned markup includes the dropdown, its options, supporting hidden controls, and JavaScript used for initial synchronization.

`GetHtml()` does not register native Vectric fields or Gadget Factory event handlers.

Example

```lua
local Preview = GF.Combobox.New("Preview")

Preview:PrepareContent({
    {"Small",  1},
    {"Medium", 2},
    {"Large",  3}
}, 20, 50, 240)

local html = Preview:GetHtml()
```

The resulting string can be used by a visual designer or other application that needs the component's HTML representation.

However, it does not by itself establish a fully functional Vectric-bound Combobox.

Relationship to WithContent

`WithContent()` begins by invoking `PrepareContent()`.

It then registers the required callbacks and component binding before adding the generated HTML to Gadget Factory's document.

Thus:

PrepareContent defines the component. GetHtml renders it. WithContent installs it.

The same architectural pattern appears throughout the Gadget Factory component library.

How Combobox works internally

Combobox differs substantially from the native Vectric dropdown implementation.

Rather than using Vectric's `AddDropDownList()` and associated item-population methods, it combines:

- A standard HTML `<select>` element.
- A hidden native-bound TextField.
- Hidden LuaButtons for event dispatch and document readiness.
- A JavaScript bridge connecting the visible dropdown to the hidden field.
- A Lua collection of item definitions and stable identifiers.

This arrangement gives Combobox control over the visible items while retaining native Vectric access to the selected state.

Automatic identifiers

The first Combobox instance receives an identifier equivalent to:

```lua
cbx1
```

Its surrounding container uses:

```lua
divcbx1
```

Additional internal controls use identifiers resembling:

```lua
selection_cbx1
change_cbx1
ready_cbx1
```

Individual items receive unique tokens such as:

```lua
cbi1_cbx1
cbi2_cbx1
```

These identifiers are generated automatically.

The programmer normally works with item indices and associated Lua values rather than the internal tokens.

Generated HTML

A simplified two-item Combobox produces markup resembling:

```HTML
<div id="divcbx1" class="cssCBX"
     style="left:20px;top:50px;width:240px;">

    <h2>Material</h2>

    <select id="cbx1">
        <option value="cbi1_cbx1">MDF</option>
        <option value="cbi2_cbx1">Plywood</option>
    </select>

    <input type="text" id="selection_cbx1"
           style="display:none;">

    <input type="button" id="change_cbx1"
           class="LuaButton" style="display:none;">

    <input type="button" id="ready_cbx1"
           class="LuaButton" style="display:none;">
</div>
```

The actual markup also includes an `onchange` handler on the `<select>`, accessibility attributes, and a JavaScript initialization block.

The internal controls are not visible to the user.

Why use item tokens?

HTML dropdown option values are strings.

Lua values, however, can be numbers, Booleans, tables, or other types.

Combobox avoids directly converting programmer-defined values into HTML option values.

Instead, each item receives a unique token.

For example:

```LUA
{"Medium", 0.03}
```

might be represented in HTML as:

```HTML
<option value="cbi2_cbx1">Medium</option>
```

The token identifies the item, while Lua retains the numeric value `0.03`.

This provides several advantages:

- Lua values do not need to be converted into strings and back again.
- Two distinct items may share the same associated value.
- Items can move to different positions without losing their identities.

- Selection can survive certain label changes and collection replacements.
- Boolean `false` values remain distinguishable from missing values.

Native binding

Combobox binds only its hidden selection field through Vectric.

The implementation is:

```lua
function Combobox:Bind(dlg)
    self.dlg, self.ready = dlg, false
    dlg:AddTextField(self.selectionId, Token(self))
end
```

The visible HTML `<select>` does not receive a native Vectric dropdown binding.

Instead, the hidden TextField holds the token identifying the selected item.

This is the central architectural distinction between Combobox and Vectric's native dropdown.

Selection changes

When the user selects an option, the dropdown's JavaScript performs two operations:

1. Copies the selected option's token into the hidden TextField.
2. Activates the hidden LuaButton associated with selection changes.

The Gadget Factory dispatcher then invokes Combobox's registered Lua handler.

The handler reads the token through Vectric's native TextField getter, finds the corresponding item, and updates the selected index.

If the selection has changed and a programmer-defined callback exists, it invokes that callback with the associated Lua value.

The event path is therefore:

```text
User changes HTML dropdown
             |
             v
JavaScript copies selected token
             |
             v
Hidden LuaButton is activated
             |
             v
Gadget Factory dispatcher
             |
             v
Vectric GetTextField retrieves token
             |
```

TEXT / CONTINUED

```
              v
Combobox resolves selected item
              |
              v
Programmer's onselect callback
```

This combines HTML's dropdown functionality with Gadget Factory's existing callback machinery.

Refreshing the item collection

When Combobox needs to update its displayed items, it uses:

LUA

```lua
self.dlg:SetInnerHtml(self.id, OptionsHtml(self))
self.dlg:UpdateTextField(self.selectionId, Token(self))
```

The first operation replaces the options inside the existing `<select>` element.

The second synchronizes the hidden Vectric-bound TextField with the current selected token.

The surrounding dropdown element and its existing binding infrastructure are preserved.

This is the same general approach discussed in Dynamic HTML (p. 364): replace the necessary HTML content while maintaining stable identifiers for the controls that continue to participate in native bindings and event dispatching.

The refresh operation also suppresses selection callbacks while programmatic changes are being applied.

Document readiness

Combobox contains its own post-load initialization mechanism.

During native binding, the dialog object exists, but Vectric's embedded browser may not yet have loaded the document.

The component therefore uses a hidden LuaButton triggered after the page loads.

That callback marks Combobox as ready and synchronizes the displayed options with the current Lua selection state.

This also accommodates changes made after native binding but before the document becomes available.

Preventing accidental first-item selection

Ordinary HTML dropdowns tend to select their first option when no option has been explicitly selected.

Combobox deliberately supports index `0`, so that browser behavior would be undesirable.

Its generated JavaScript restores the intended selection using the options' `defaultSelected` properties.

It uses `MutationObserver` when available, with a Trident-compatible fallback based on `onpropertychange`.

This ensures that rebuilding the option collection does not inadvertently change an intentionally empty selection into the first available choice.

Cleanup

Combobox implements a `Done()` method that Gadget Factory invokes during its normal component cleanup.

The method reads the current selection, releases the stored dialog reference, and marks the component as no longer ready.

Its selected value and index remain available through the instance's stored state.

This is useful when retrieving the selected choice during processing after the dialog has closed.

Styling

The supplied `Combobox.css` stylesheet controls the dropdown's positioning and appearance.

Container

The outer container uses the `cssCBX` class.

```css
.cssCBX {
    position: absolute;
    box-sizing: border-box;
    padding: 6px 10px 15px 18px;
}
```

The component's `WithContent()` arguments supply the container's left, top, and width values.

Dropdown appearance

The visible HTML `<select>` occupies the width of its containing element and has a height of 32 pixels.

```css
.cssCBX select {
    display: block;
    box-sizing: border-box;
    width: 100%;
    height: 32px;
    padding: 3px 6px;
    font-family: "Segoe UI", sans-serif;
    font-size: 16px;
    background-color: #f0fdff;
    border: 1px solid #4b5ba8;
}
```

The stylesheet also defines hover, focus, disabled, and option appearances.

Custom dropdown arrow

The stylesheet suppresses Trident's native dropdown arrow:

```css
.cssCBX select::-ms-expand {
    display: none;
}
```

It replaces that arrow with CSS-generated decorative elements.

The arrow itself uses a character from the `Wingdings 3` font:

```css
.cssCBX::after {
    content: "q";
    font-family: "Wingdings 3";
}
```

A second pseudo-element supplies its background and border.

Both decorative elements use `pointer-events:none` so that they do not interfere with normal dropdown interaction.

This approach allows Combobox to maintain a consistent appearance in the embedded browser.

It also depends on the specified font being available.

Gadget-specific styling

A gadget can override the component's styles through AddStyle (p. 108).

For example:

```lua
GF.AddStyle([[
    .cssCBX select {
        background-color: #ffffff;
        border-color: #888888;
    }
]])
```

As with other components, styling overrides should account for CSS specificity and browser compatibility.

Combobox versus RadioGroup

Combobox and RadioGroup both associate displayed choices with programmer-defined Lua values.

Their primary differences are presentation and runtime flexibility.

Feature	Combobox	RadioGroup
Presentation	Dropdown	Visible group of radio buttons
Selection model	One item or none	One selected item

Feature	Combobox	RadioGroup
Item definition	`{label, value}`	`{label, value}`
Programmatic selection	`Select(index)`	`Select(index)`
Selection retrieval	`Selected()`	`Selected()`
Clear selection	`Select(0)`	No dedicated clearing operation
Runtime item modification	Supported	Not provided by the public API
Selection callback	`(value, combo)`	`(value, group)`
Native binding	Hidden TextField	Vectric RadioGroup

Use RadioGroup when the available choices should remain visible and can reasonably fit within the interface.

Use Combobox when space is limited, the item collection may change, or an explicit unselected state is useful.

Important considerations

- **Library loading:** Load `Combobox` before creating instances.
- **Multiple instances:** Each instance receives unique generated identifiers and item tokens.
- **Item labels:** Labels must be strings.
- **Associated values:** Lua values are stored separately from HTML option tokens. Values may be duplicated and may include `false` or `nil`.
- **One-based indexing:** Defined items use indices beginning with `1`. Index `0` represents no selection.
- **No automatic selection:** Without an explicit selection, Combobox begins with no selected item.
- **Optional blank choice:** `AllowEmptySelection()` provides a visible way for the user to choose no item.
- **Index validation:** Invalid, noninteger, or out-of-range selection indices cause assertion errors.
- **Runtime modification:** `SetItems()`, `AddItem()`, `RemoveItem()`, `ReplaceItem()`, and `Clear()` support programmatic item management.
- **Selection preservation:** `SetItems()` attempts to match existing items by label and value, then by non-`nil` value.
- **Stable identity:** Changing an item's label while retaining its non-`nil` value can preserve the selection.
- **Programmatic callbacks:** Programmatic selection and item-management methods do not explicitly invoke `onselect`.
- **Selection callbacks:** User changes supply the associated value and originating instance.
- **Read-only rendering:** `PrepareContent()` and `GetHtml()` do not register native bindings or event handlers.
- **Single installation:** `WithContent()` can be called only once per instance.

- **Native binding:** The component uses a hidden Vectric TextField rather than Vectric's native dropdown binding.
- **Document readiness:** Combobox synchronizes its state after the browser loads the dialog.
- **Safe text rendering:** Titles, item labels, and relevant HTML attribute values are escaped before insertion into generated markup.
- **Runtime state:** `Selected()` reads native state while the component is ready; `SelectedValue()` uses the currently stored selected index.
- **Styling:** The dropdown uses Trident-specific CSS and a custom arrow based on Wingdings 3.
- **No automatic persistence:** Combobox does not save its items or selection between independent gadget executions.
- **Not a direct native dropdown wrapper:** Methods such as `GetDropDownListValue()` are not used to retrieve its selection.

Related functions

- LoadComponents (p. 36) - Load Gadget Factory component libraries.
- RadioGroup (p. 120) - Create mutually exclusive visible selections.
- CheckboxGroup (p. 137) - Manage multiple independent Boolean options.
- ListBox (p. 217) - Create a richer list-selection control.
- Dynamic HTML (p. 364) - Understand runtime HTML replacement and preserved identifiers.
- Event Handling (p. 320) - Understand the LuaButton dispatcher.
- AddBinding (p. 329) - Understand Vectric's native field-registration mechanism.
- InitializeScript (p. 34) - Construct and register components.

Summary

`GF.Combobox` provides a compact dropdown control with programmer-defined Lua values and a dynamically editable collection of choices.

Its public API includes:

Method	Purpose
`New()`	Create a Combobox instance.
`WithContent()`	Prepare and install the dropdown.
`AllowEmptySelection()`	Enable a visible blank choice.
`Select()`	Set or clear the selection by index.

Method	Purpose
`Selected()`	Retrieve the selected value and index.
`SelectedValue()`	Retrieve the stored selected value.
`SetItems()`	Replace the item collection.
`AddItem()`	Insert or append a choice.
`RemoveItem()`	Remove a choice.
`ReplaceItem()`	Replace an existing choice.
`Clear()`	Remove every choice.
`GetItems()`	Retrieve the current item definitions.
`Count()`	Return the number of defined items.
`PrepareContent()`	Prepare the dropdown without installation.
`GetHtml()`	Generate its HTML independently.

Combobox combines a familiar HTML dropdown with Gadget Factory's Lua event dispatching and Vectric's native TextField binding.

The key design decision is that **HTML manages the choices, Lua manages their identities and values, and Vectric transports the selected item between them**.

This arrangement provides dynamic item management without depending on the limitations of Vectric's native dropdown-population API.

RadioGroup presents the choices. Combobox keeps them compact - and lets the programmer change them.

REFERENCE 5.7

ListBox

`GF.ListBox` creates a scrollable, single-selection list control with programmer-defined items and associated Lua values.

Although it presents the appearance and behavior of a conventional listbox, it is implemented using a collection of vertically stacked HTML buttons rather than a native Vectric list control.

ListBox provides programmatic selection, keyboard navigation, optional selection callbacks, and methods for modifying the available items while the dialog is running.

Each item consists of a descriptive label and an associated Lua value.

The component manages its HTML structure, selection state, native Vectric binding, and event dispatching automatically.

Like Combobox (p. 192), ListBox supports dynamically editable item collections, but displays its choices simultaneously within a scrollable region rather than a dropdown.

Loading the component

ListBox is supplied as a separate library module and must be loaded before use.

```lua
GF.LoadComponents {"ListBox"}
```

This exposes the `GF.ListBox` constructor and includes the associated `ListBox.css` stylesheet.

The module is loaded once per gadget. Multiple independent ListBox instances may then be created.

Basic usage

A ListBox is created using `New()`, populated using `WithContent()`, and optionally given an initial selection using `Select()`.

```lua
local Layers = GF.ListBox.New("Destination Layer")
    :WithContent({
        {"Cutting",   "Cutting"},
        {"Engraving", "Engraving"},
        {"Pockets",   "Pockets"},
        {"Drilling",  "Drilling"}
    }, 30, 60, 240, 160)
    :Select(2)
```

This creates a ListBox titled **Destination Layer**, positioned 30 pixels from the left and 60 pixels from the top.

The list is 240 pixels wide and 160 pixels high.

The second item, Engraving, is initially selected.

Each item follows the familiar convention:

```lua
{label, value}
```

The label identifies the choice to the user, while the associated value is returned to the gadget.

Construction sequence

The normal sequence consists of three operations:

1. `New()` creates the ListBox instance and establishes its optional callback.
2. `WithContent()` prepares the items, generates HTML, and registers the required native binding and event handlers.
3. `Select()` establishes an initial selection, if desired.

The methods return the ListBox instance, allowing the fluent construction style demonstrated above.

Unlike RadioGroup (p. 120), ListBox does not automatically select the first item.

Its default selection index is `0`, representing no selection.

New

`ListBox.New()` creates a new ListBox instance.

Syntax

```lua
local list = GF.ListBox.New(title, onselect)
```

Parameters

Parameter	Description
`title`	Optional heading displayed above the list.
`onselect`	Optional Lua function invoked when the user changes the selection.

Return value: A new ListBox instance.

Without a callback

```lua
local Layers = GF.ListBox.New("Destination Layer")
```

The list operates normally, but user selections do not invoke a programmer-defined callback.

The selected value can be retrieved later using `Selected()`.

This is appropriate when the selection is needed only during final processing.

With a callback

```lua
local LayerChanged = function(value, list)
    SelectedLayer = value
end

local Layers = GF.ListBox.New("Destination Layer", LayerChanged)
```

The callback receives two arguments:

Argument	Description
`value`	Lua value associated with the newly selected item.
`list`	ListBox instance that generated the event.

The originating instance is useful when several ListBoxes share the same callback.

The constructor validates the callback argument. It must be a function or `nil`.

Optional title

When a nonempty title is supplied, ListBox generates a heading positioned just above the scrollable area.

If the title is omitted or empty, no heading is generated.

The list retains its normal functionality.

WithContent

`WithContent()` establishes the items, dimensions, position, and event-handling infrastructure.

Syntax

```lua
list:WithContent(items, left, top, width, height)
```

Parameters

Parameter	Description
`items`	Array-style Lua table containing `{label, value}` pairs.
`left`	Horizontal position in pixels. Defaults to `0`.
`top`	Vertical position in pixels. Defaults to `0`.
`width`	Width in pixels. Defaults to `200`.
`height`	Height in pixels. Defaults to `160`.

Return value: The ListBox instance.

Item structure

Every item uses the following convention:

```lua
{label, value}
```

For example:

```lua
local Materials = GF.ListBox.New("Material")
    :WithContent({
        {"MDF",       "mdf"},
        {"Plywood",   "plywood"},
        {"Hardwood", "hardwood"}
    }, 20, 60, 250, 180)
```

The displayed labels are independent of the values returned to Lua.

Selecting Hardwood produces `"hardwood"`.

Supported value types

The item label must be a string.

The associated value is retained in Lua and may be a string, number, Boolean, table, or another Lua value.

For example:

```lua
local Depths = GF.ListBox.New("Pass Depth")
    :WithContent({
        {"Shallow", 0.02},
        {"Medium",  0.03},
        {"Deep",    0.05}
    }, 20, 60, 220, 150)
```

The selected numeric value is returned without requiring string-to-number conversion.

Associated values may also be `false` or `nil`.

> **Important:** When an item can have a `nil` or `false` value, use the selected index to determine whether a selection exists rather than testing the returned value's truthiness.

Item validation

ListBox requires a proper Lua sequence of item definitions.

The item table must use consecutive positive integer indices.

Each item must be a table whose first element is a string label.

Invalid item structures cause an assertion error.

Empty item collections are permitted:

```lua
list:WithContent({}, 20, 60, 250, 180)
```

This is useful when the choices will be populated later.

Geometry validation

All geometry arguments must be finite numbers.

Width and height must be greater than zero.

Whole-pixel dimensions are conventional, but the implementation does not require integer values.

Registration

`WithContent()` performs several operations:

1. Prepares the item collection and internal identifiers.
2. Registers the component's document-ready handler.
3. Registers the selection-change handler.
4. Generates the ListBox HTML.
5. Registers the component with Gadget Factory for native binding.

Unlike ordinary button groups, ListBox registers an actual native Vectric field.

`WithContent()` may be called only once per instance.

A second call raises an assertion error.

Use the item-management methods to change a list after installation.

Select

`Select()` establishes or changes the selected item by its one-based index.

Syntax

```lua
list:Select(index)
```

Parameters

Parameter	Description
`index`	One-based item index, or `0` for no selection.

Return value: The ListBox instance.

Selecting an item

```lua
Layers:Select(3)
```

Selects the third item in the collection.

The index identifies the item's position, not its associated value.

For example:

```lua
local Depths = GF.ListBox.New("Pass Depth")
    :WithContent({
        {"Shallow", 0.02},
        {"Medium",  0.03},
        {"Deep",    0.05}
    }, 20, 60, 220, 150)

Depths:Select(2)
```

Selects Medium, whose associated value is `0.03`.

Clearing the selection

ListBox explicitly supports an unselected state.

```lua
Layers:Select(0)
```

Calling `Select()` without an argument has the same effect:

```lua
Layers:Select()
```

This clears the selection without removing any items.

The current ListBox has no special visible Clear Selection item. Once the user selects a row, clicking that row again does not deselect it.

Clearing is normally performed programmatically.

Invalid indices

ListBox validates selection indices.

An index must be an integer between `0` and the current item count.

For example:

```lua
Layers:Select(-1)
Layers:Select(99)
```

Both calls raise an assertion error when those indices are outside the permitted range.

Programmatic updates

Once the dialog is ready, `Select()` updates the displayed selection.

It does not explicitly invoke the programmer's `onselect` callback.

This distinction separates programmatic state changes from user interaction.

Selected

`Selected()` retrieves the current selection.

Syntax

```lua
local value, index = list:Selected()
```

Return values

Return	Description
`value`	Lua value associated with the selected item.
`index`	One-based selected index, or `0` if nothing is selected.

Example

Given:

```lua
Layers:Select(2)
```

The following call:

```lua
local value, index = Layers:Selected()
```

returns:

```lua
-- value = "Engraving"
-- index = 2
```

Retrieving only the value

```lua
local layer = Layers:Selected()
```

Lua retains only the first return value.

Retrieving only the index

```lua
local _, index = Layers:Selected()
```

The underscore is simply a conventional variable name indicating that the first result is not needed.

No selection

When the list has no selected item:

```lua
local value, index = Layers:Selected()

-- value = nil
-- index = 0
```

An item whose associated value is `nil` also produces a `nil` first result.

However, its index remains nonzero.

This makes the index the reliable indicator of whether an item is selected.

Native state synchronization

While the dialog is ready, `Selected()` reads the current selected-item token from Vectric.

It then resolves that token to the corresponding Lua item.

Consequently, the method reflects the current dialog selection rather than merely returning the originally configured index.

After the dialog closes, the component's cleanup mechanism retains the final selection in its stored state.

This allows the selected value to remain available to application processing.

Managing items dynamically

ListBox provides a complete set of item-management methods.

These allow the programmer to alter the available choices without recreating the dialog or installing additional native fields.

Method	Purpose
`SetItems()`	Replace the entire item collection.
`AddItem()`	Insert or append an item.
`RemoveItem()`	Remove an item by index.
`ReplaceItem()`	Replace an existing item.
`Clear()`	Remove all items.
`GetItems()`	Return a copy of the item definitions.
`Count()`	Report the number of items.

With the exception of `GetItems()` and `Count()`, these methods return the ListBox instance.

Changes are immediately reflected in the visible list when the dialog is ready.

Before that point, they modify the prepared Lua state that will be displayed when initialization completes.

SetItems

`SetItems()` replaces the entire item collection.

```lua
list:SetItems(items)
```

Return value: The ListBox instance.

Example:

```lua
Layers:SetItems({
    {"Roughing",  "Roughing"},
    {"Finishing", "Finishing"},
    {"Drilling",  "Drilling"}
})
```

The old choices are replaced by the new collection.

Preserving selection

ListBox attempts to preserve the identity of existing items when their definitions change.

Its matching process has two stages:

1. Match new items against old items using both label and value.
2. Match remaining items using their non-`nil` associated values.

Consider:

```lua
Layers:SetItems({
    {"Cutting",   "cut"},
    {"Engraving", "engrave"},
    {"Pockets",   "pocket"}
})
```

If Engraving is selected, replacing the collection with:

```lua
Layers:SetItems({
    {"Pockets",        "pocket"},
    {"Detailed Engraving", "engrave"},
    {"Cutting",        "cut"}
})
```

preserves the selected item.

Its index may change, but the associated value identifies it as the same logical choice.

Duplicate values

Several items may share the same associated value.

ListBox preserves their identities using stable internal tokens.

When matching repeated values, it processes their occurrences in order rather than treating every matching value as the same item.

This permits duplicate values without automatically collapsing the corresponding choices into one identity.

Nil-valued items

For items whose associated value is `nil`, an exact label-and-value match is required to preserve identity.

Changing the label of a selected nil-valued item can therefore clear the selection.

When selection persistence matters, a stable non-`nil` value is preferable.

AddItem

`AddItem()` inserts or appends a new item.

```lua
list:AddItem(pair, index)
```

Parameter	Description
`pair`	New `{label, value}` item.
`index`	Optional insertion position, defaulting to the end.

Return value: The ListBox instance.

Append:

```lua
Layers:AddItem({"New Layer", "new"})
```

Insert at position two:

```lua
Layers:AddItem({"Construction", "construction"}, 2)
```

If a selected item is shifted by the insertion, ListBox adjusts its stored index so that the same item remains selected.

Valid insertion indices range from `1` through `Count() + 1`.

RemoveItem

`RemoveItem()` removes an item by index.

```lua
list:RemoveItem(index)
```

Return value: The ListBox instance.

Example:

```lua
Layers:RemoveItem(2)
```

If an earlier item is removed, the selected index is adjusted to preserve the current selection.

If the selected item itself is removed, the selection is cleared.

Invalid indices cause an assertion error.

ReplaceItem

`ReplaceItem()` replaces one existing item.

```lua
list:ReplaceItem(index, pair)
```

Return value: The ListBox instance.

For example:

```lua
Layers:ReplaceItem(2, {
    "Fine Engraving", "engrave"
})
```

When the associated non-`nil` value remains unchanged, changing only the displayed label preserves the item's identity and selection.

When the associated value changes, ListBox treats the replacement as a different item.

If the original item was selected, that selection is cleared.

For nil-valued items, both label and value must remain unchanged to preserve their identity.

Clear

`Clear()` removes all items.

```lua
list:Clear()
```

Return value: The ListBox instance.

This is equivalent to:

```lua
list:SetItems({})
```

The selected index becomes `0`.

The ListBox container remains available and may be populated again later.

GetItems

`GetItems()` returns the current item definitions.

```lua
local items = Layers:GetItems()
```

Return value: A new array containing `{label, value}` pairs.

The returned array and its item-pair tables are copies.

For example:

```lua
local items = Layers:GetItems()

items[#items + 1] = {"New Layer", "new"}

Layers:SetItems(items)
```

This retrieves the existing items, appends another choice, and replaces the component's collection.

Changes made only to the returned array do not modify the ListBox.

However, associated values that are themselves tables or other reference types are not deep-copied.

Count

`Count()` reports the number of defined items.

```lua
local count = Layers:Count()
```

Return value: Integer representing the current item count.

It may be called before or after the dialog is created because it reads the component's Lua item collection.

Selection callbacks

ListBox optionally invokes a programmer-defined callback when the user changes the selected item.

Example

```lua
local SelectedLayer

local LayerChanged = function(value, list)
    SelectedLayer = value
end

local Layers = GF.ListBox.New("Destination", LayerChanged)
    :WithContent({
        {"Cutting",   "Cutting"},
        {"Engraving", "Engraving"},
        {"Pockets",   "Pockets"}
    }, 30, 60, 240, 180)
```

When the user selects Engraving, ListBox effectively invokes:

```lua
LayerChanged("Engraving", Layers)
```

The callback receives the associated Lua value and the originating ListBox instance.

Shared callbacks

Several ListBoxes may use one callback function.

```lua
local SelectionChanged = function(value, list)
    if list == SourceList then
        SourceValue = value
    elseif list == DestinationList then
        DestinationValue = value
    end
end
```

The instance argument identifies which list generated the event.

Callback behavior

A callback is invoked when a user action changes the selected item.

ListBox identifies the change by comparing its stable item tokens.

Consequently, selecting a different item with the same associated Lua value still counts as a change.

Programmatic operations such as `Select()` and `SetItems()` do not explicitly invoke `onselect`.

The component also suppresses its selection-change processing while programmatically refreshing the displayed rows.

Without a callback

When no `onselect` function is supplied, ListBox still maintains its internal selection.

Its hidden event-handling mechanism remains necessary to synchronize the displayed selection with Vectric.

However, no programmer-defined selection callback is invoked.

This is an important distinction from CheckboxGroup and RadioGroup, where optional callbacks determine whether individual HTML inputs require LuaButton handling.

Keyboard navigation and accessibility

Although ListBox is constructed from buttons, it provides navigation behavior expected of a list-selection control.

Keyboard navigation

The component handles the following keys:

Key	Behavior
Up Arrow	Select the previous item.
Down Arrow	Select the next item.
Home	Select the first item.
End	Select the last item.

The selected row receives keyboard focus when one of these navigation operations is performed.

Navigation remains within the first and last items.

Ordinary button activation also applies to the individual rows.

ARIA attributes

ListBox uses Accessible Rich Internet Applications (ARIA) attributes to communicate its list-selection structure.

The outer list container uses:

```
CSS
role="listbox"
```

Each item uses:

```
CSS
role="option"
```

The current selection is represented through:

```
CSS
aria-selected="true"
```

or:

```
CSS
aria-selected="false"
```

These attributes also serve as styling hooks.

Keyboard focus

Each item has a `tabindex` value.

The selected item receives:

```lua
tabindex="0"
```

Other items receive:

```lua
tabindex="-1"
```

When no item is selected, the first row receives the normal tab stop.

This helps keyboard navigation treat the list as a coordinated control rather than an unrelated collection of buttons.

The component implements these specific behaviors; it should not be assumed to provide every feature of a fully standardized accessibility widget.

Example: A destination-layer selector

A ListBox is particularly useful when the user must select from a collection of layers or other named objects.

The following example demonstrates selection and validation.

```lua
local DestinationList
local SelectedDestination

local DestinationChanged = function(value, list)
    SelectedDestination = value

    -- Permit execution only when a destination is selected
    local _, index = list:Selected()
    GF.ExecutionEnabled(index > 0)
end

local function InitializeScript(job)
    GF.LoadComponents {"ListBox"}

    DestinationList = GF.ListBox.New(
        "Destination Layer",
        DestinationChanged
    ):WithContent({
        {"Cutting",   "Cutting"},
        {"Engraving", "Engraving"},
        {"Pockets",   "Pockets"}
    }, 30, 60, 240, 180)
end
```

The user selects the destination directly from the visible list.

The callback stores the associated value and enables the standard Execute button.

The example assumes that other portions of the gadget provide the necessary processing and lifecycle definitions.

Dynamically refreshing the choices

Suppose the available destinations change.

```lua
local function RefreshDestinations(items)
    DestinationList:SetItems(items)

    local value = DestinationList:Selected()
    DestinationChanged(value, DestinationList)
end

RefreshDestinations({
    {"New Cutting Layer", "Cutting"},
    {"Engraving",         "Engraving"},
    {"Finishing",         "Finishing"}
})
```

ListBox rebuilds the visible rows while attempting to preserve the selected item's identity.

If Pockets was selected, this new collection removes it and clears the selection. `SetItems()` does not invoke the selection callback, so `RefreshDestinations()` explicitly calls `DestinationChanged()` afterward. That keeps `SelectedDestination` and the Execute button consistent with the list's actual selection.

Use this helper when replacing the collection. After other programmatic selection or item changes, perform the same resynchronization if application state depends on the selection callback.

The gadget does not need to recreate the native dialog, reload the component, or register a new field.

For a real layer-selection gadget, the item collection could be constructed from the current Vectric job.

Preparing and rendering content independently

ListBox separates preparation and HTML generation from installation, following the same pattern as the other components.

PrepareContent

```lua
list:PrepareContent(items, left, top, width, height)
```

The arguments are the same as those accepted by `WithContent()`.

Parameter	Description	Default
`items`	An array-style sequence of `{label, value}` pairs.	Required
`left`	Horizontal position in pixels.	`0`
`top`	Vertical position in pixels.	`0`

Parameter	Description	Default
`width`	Container width in pixels.	`200`
`height`	Container height in pixels.	`160`

Labels must be strings. Associated values may include duplicates, `false`, or `nil`. Geometry values must be finite numbers, and width and height must be greater than zero.

Return value: The ListBox instance, allowing further calls to be chained.

The method validates the geometry, calls `SetItems()` to prepare the collection, and stores the layout properties. Existing item identities and selections follow the matching rules described under SetItems (p. 225).

It does not register event handlers, queue native bindings, or add HTML to Gadget Factory's document.

On an already live ListBox, preparation can also change the visible rows. Once the instance is ready, `SetItems()` refreshes its content in the browser. Layout properties are stored afterward; `PrepareContent()` is not a supported way to move or resize a live control.

Use a separate, uninstalled instance when preparing HTML for a preview.

GetHtml

```lua
local html = list:GetHtml()
```

Return value: A string containing the complete HTML for the prepared ListBox.

The returned markup includes:

- The positioned outer container and optional title.
- The list region and its item buttons.
- A hidden TextField carrying the selected item's token.
- Hidden LuaButtons for selection changes and document readiness.
- JavaScript for selection, keyboard navigation, and the readiness callback.

`GetHtml()` uses the instance's prepared content and layout. It does not add the string to the dialog, register native fields, or install Gadget Factory event handlers.

The supporting controls and JavaScript are part of the returned string. Generating that string alone does not establish a working Vectric-bound ListBox.

Example

With the ListBox library already loaded:

```lua
local Preview = GF.ListBox.New("Preview")

local html = Preview:PrepareContent({
    {"Cutting",   "Cutting"},
    {"Engraving", "Engraving"},
    {"Pockets",   "Pockets"}
}, 20, 50, 240, 160):GetHtml()
```

A visual designer can use `html` to inspect or display the component's appearance without installing `Preview` as a working control.

Relationship to WithContent

`WithContent()` begins by calling `PrepareContent()`. It then registers the readiness and selection callbacks, queues the component for native binding, and adds the generated HTML to Gadget Factory's document.

For normal gadget construction, use `WithContent()`. Use `PrepareContent()` and `GetHtml()` when the HTML representation itself is what you need.

How ListBox works internally

This is where the implementation becomes particularly interesting.

ListBox is not a native Vectric list control. It is a collection of HTML buttons that collectively behave as one.

The mechanism relies on four facilities:

1. HTML buttons represent the visible rows.
2. CSS gives those buttons the appearance of list entries.
3. JavaScript manages interaction and selection.
4. A hidden Vectric-bound TextField communicates the selection to Lua.

Gadget Factory's existing event dispatcher connects the JavaScript interaction to the component's Lua code.

Automatic identifiers

The first ListBox instance receives an identifier equivalent to:

```lua
lb1
```

Its outer container uses:

```lua
divlb1
```

Additional internal controls receive names resembling:

```lua
selection_lb1
change_lb1
ready_lb1
```

Each list item receives a stable, automatically generated token:

```lua
lbi1_lb1
lbi2_lb1
lbi3_lb1
```

These tokens identify individual items independently of their positions or associated Lua values.

The programmer does not normally need to manage them.

Generated HTML

A simplified ListBox produces markup resembling:

```html
<div id="divlb1" class="ListBox"
     style="left:30px;top:60px;width:240px;height:160px;">

    <h2 class="ListBox-title">Destination Layer</h2>

    <div id="lb1" class="ListBox-list"
         role="listbox" aria-label="Destination Layer">

        <input type="button" class="ListBox-item"
               id="lbi1_lb1" role="option"
               aria-selected="false" tabindex="0"
               value="Cutting" title="Cutting">

        <input type="button" class="ListBox-item"
               id="lbi2_lb1" role="option"
               aria-selected="false" tabindex="-1"
               value="Engraving" title="Engraving">
    </div>

    <input type="text" id="selection_lb1"
           style="display:none;">

    <input type="button" class="LuaButton"
           id="change_lb1" style="display:none;">

    <input type="button" class="LuaButton"
           id="ready_lb1" style="display:none;">
</div>
```

The actual markup also contains JavaScript for mouse selection, keyboard navigation, and document readiness.

Notice that the visible rows use:

```css
class="ListBox-item"
```

They do **not** use:

```css
class="LuaButton"
```

This is deliberate.

A separate hidden LuaButton reports changes for the entire list.

One event handler for every row

Rather than registering an individual Lua callback for every button, ListBox attaches one JavaScript click handler to the list container.

The essential logic is:

```javascript
list.onclick = function(event) {
    var item = (event || window.event).target
        || window.event.srcElement;

    if (item.parentNode !== list || item.type !== 'button')
        return;

    // Update the selected row's appearance

    field.value = item.id;

    document.getElementById(changeId).click();
};
```

Here, `changeId` represents the generated ID of the hidden change button.

The actual module inserts that identifier while generating its JavaScript.

The handler receives events from the individual rows and determines which button was activated.

It then updates the selected-row attributes, copies the item's token into the hidden TextField, and activates the hidden LuaButton.

This technique is called **event delegation**.

Why event delegation matters

JavaScript events from child elements can be handled by their parent container.

This means the list requires only one event handler, regardless of how many rows it contains.

More importantly, when ListBox replaces its rows, the parent container and its event handler remain intact.

Newly inserted buttons immediately participate in the existing interaction mechanism.

No new Lua handlers are required when items are added or removed.

This is one of the most useful architectural features of the component.

The native Vectric binding

ListBox registers exactly one native Vectric field to transport its selected-item token.

Its binding method is:

```lua
function ListBox:Bind(dlg)
    self.dlg, self.ready = dlg, false

    dlg:AddTextField(
        self.selectionId,
        Token(self)
    )
end
```

The native binding is a hidden TextField.

Vectric does not need to know how many items exist or how they are displayed.

It only needs to transport the identifier of the currently selected item.

Selection event path

The complete interaction sequence is:

```text
User clicks a visible list row
             |
             v
JavaScript identifies the button
             |
             v
Selected-row attributes are updated
             |
             v
Button token is copied to hidden TextField
             |
             v
Hidden LuaButton is activated
             |
             v
Gadget Factory dispatches the event
             |
             v
ListBox reads the token from Vectric
             |
             v
Token is resolved to a Lua item
             |
             v
Optional onselect callback executes
```

The underlying mechanisms are familiar from Event Handling (p. 320), AddBinding (p. 329), and Dynamic HTML (p. 364).

ListBox combines them into a control whose implementation is largely invisible to the programmer.

Stable item identity

Each item receives a unique token when first created.

The component retains those tokens whenever an item can be recognized across changes to the collection.

For example, a selected item may move to a different index without changing its underlying identity.

This is why `SetItems()` and `ReplaceItem()` can preserve selections even when the displayed labels or item positions change.

The token belongs to the logical item, not simply to its current position.

Refreshing the displayed rows

ListBox updates its contents using:

```lua
self.dlg:SetInnerHtml(self.id, ItemsHtml(self))

self.dlg:UpdateTextField(
    self.selectionId,
    Token(self)
)
```

The first operation replaces the visible button rows.

The second synchronizes the hidden Vectric-bound field with the selected item's token.

The surrounding list container is not replaced.

Consequently, its JavaScript event handlers remain available for the new rows.

During this operation, the component temporarily sets an internal updating flag to prevent programmatic replacement from being processed as a user-generated selection change.

This is another practical application of the stable-container principle discussed in Dynamic HTML (p. 364).

Document readiness

ListBox also manages the distinction between native dialog creation and the browser's document becoming available.

Its generated JavaScript installs a document-load handler that activates a hidden ready button.

The Gadget Factory callback then marks the component as ready and refreshes its displayed items.

This ensures that selections or item changes established before the dialog opens are synchronized once the document is usable.

It also avoids attempting dynamic HTML replacement before the embedded browser is ready.

Cleanup

ListBox implements a `Done()` method that Gadget Factory invokes during component cleanup.

The method retrieves the current selection before releasing its native dialog reference.

Its stored selection index remains available afterward.

This allows the component to retain the final selected value without keeping the native dialog object alive.

Styling

The supplied `ListBox.css` stylesheet is responsible for making the stacked buttons resemble a conventional listbox.

Outer container

The main container uses:

```css
.ListBox {
    position: absolute;
    z-index: 0;
}
```

Its Left, Top, Width, and Height are supplied by the component's generated inline styling.

Scrollable list region

The inner container uses:

```css
.ListBox-list {
    position: absolute;
    z-index: 1;
    left: 0;
    top: 0;
    width: 100%;
    height: 100%;
    overflow-y: auto;
    overflow-x: hidden;
    padding: 3px;
    box-sizing: border-box;
    background-color: #FFFFFF;
    border: 1px solid #777777;
}
```

When the number of rows exceeds the available height, the browser provides vertical scrolling.

Horizontal overflow is suppressed.

Individual rows

Every visible row is an HTML button styled with the `ListBox-item` class.

```css
.ListBox input.ListBox-item {
    display: block;
    width: 100%;
    height: 28px;
    box-sizing: border-box;
    margin: 0;
    padding: 2px 6px;
    background-image: none;
    background-color: white;
    border: 1px solid transparent;
    border-radius: 0;
    box-shadow: none;
    text-align: left;
    font: inherit;
    overflow: hidden;
    text-overflow: ellipsis;
    white-space: nowrap;
}
```

These rules remove the appearance of ordinary command buttons.

Each button becomes a flat, full-width list row.

Long labels are truncated with an ellipsis when they exceed the available width.

Selected appearance

The selected row is identified through its `aria-selected` attribute.

```css
.ListBox input.ListBox-item[aria-selected="true"] {
    background-color: #DDEEFF;
    border-color: cornflowerblue;
}
```

This produces the familiar light-blue selected-row appearance.

Hover and focus

The stylesheet uses a pale-yellow background for hovered rows.

```css
.ListBox input.ListBox-item:hover,
.ListBox input.ListBox-item:active {
    background-color: lemonchiffon;
}
```

Keyboard focus uses a dotted outline.

These visual states are handled through CSS rather than additional Lua callbacks.

Title positioning

The title is positioned above the list region:

```css
.ListBox-title {
    position: absolute;
    top: -26px;
    height: 24px;
    left: 4px;
    right: 0;
}
```

Because it extends above the ListBox's nominal top coordinate, the programmer should leave room for the heading when placing the component near the top of the dialog.

Gadget-specific styling

The component's appearance can be modified through AddStyle (p. 108).

For example:

```lua
GF.AddStyle([[
    .ListBox-list {
        background-color: #f8fbff;
    }

    .ListBox input.ListBox-item {
        height: 32px;
    }

    .ListBox input.ListBox-item[aria-selected="true"] {
        background-color: #cce8ff;
    }
]])
```

This changes the row height and selected appearance without modifying the ListBox module.

Because the rows are ordinary HTML buttons, their appearance is governed by the browser's CSS rendering.

ListBox versus Combobox

ListBox and Combobox share many of their item-management conventions, but they present their choices differently.

Feature	ListBox	Combobox
Presentation	Scrollable visible list	Dropdown
HTML selection control	Stacked buttons	`<select>`
Item definition	`{label, value}`	`{label, value}`
Selection model	One item or none	One item or none

Feature	ListBox	Combobox
Initial selection	Index `0`	Index `0`
Programmatic selection	`Select(index)`	`Select(index)`
Value retrieval	`Selected()`	`Selected()`
Dynamic item modification	Supported	Supported
Explicit height	Yes	No
Keyboard navigation	Arrow keys, Home, End	Native dropdown behavior
Native binding	Hidden TextField	Hidden TextField
Event bridge	Delegated button clicks	Dropdown change event
Visible empty option	Not provided	`AllowEmptySelection()`
`SelectedValue()` convenience method	Not provided	Provided

Both components maintain their own item definitions and resolve selected values through internal tokens.

Combobox conserves space. ListBox keeps the choices visible.

The appropriate choice depends primarily on the interface's layout and how frequently users need to examine or change the selection.

Important considerations

- **Library loading:** Load ListBox before constructing an instance.
- **Multiple instances:** Each ListBox receives automatically generated identifiers.
- **Single selection:** The component supports one selected item at a time, not multiple simultaneous selections.
- **Item labels:** Labels must be strings.
- **Associated values:** Values are ordinary Lua values and may include duplicates, `false`, or `nil`.
- **Selection indices:** Defined items use one-based indices; `0` represents no selection.
- **Initial selection:** No item is selected unless `Select()` establishes one.
- **Invalid indices:** Out-of-range or noninteger indices raise assertion errors.
- **Callback arguments:** The callback receives the associated value and originating instance.
- **Programmatic changes:** `Select()` and item-management methods do not explicitly invoke the selection callback.
- **Stable identities:** Internal tokens preserve item identity across supported collection changes.

- **Unknown selections:** Removing a selected item or replacing its associated value clears the selection.
- **Runtime modification:** Items can be changed without recreating the native dialog.
- **Sequence validation:** The item collection must be a proper array-style Lua sequence.
- **Native binding:** One hidden TextField transports the selected token for the entire component.
- **Delegated events:** Visible rows do not require individual LuaButton registrations.
- **Keyboard support:** Up, Down, Home, and End are handled explicitly.
- **No visible clear command:** Clearing the selection is normally performed through `Select(0)`.
- **Tooltip behavior:** Individual buttons use the string representation of their associated Lua value as the HTML `title` attribute. This may be less informative for complex values such as tables.
- **Independent rendering:** `PrepareContent()` and `GetHtml()` do not register native bindings or callback handlers. Preparation can refresh a live instance's rows; use an uninstalled instance for previews.
- **Single installation:** `WithContent()` may be called only once per instance.
- **Dynamic updates:** The list container remains stable while its button rows are replaced.
- **No automatic persistence:** Items and selections are not automatically saved between independent gadget executions.
- **HTML escaping:** Titles, labels, item values used in attributes, and other generated text are HTML-escaped.
- **Browser compatibility:** Dynamic replacement, scrolling, and keyboard navigation rely on behavior supported by Vectric's embedded Trident engine.

Related functions

- LoadComponents (p. 36) - Load Gadget Factory component libraries.
- Combobox (p. 192) - Create a compact dropdown with dynamically managed items.
- RadioGroup (p. 120) - Create mutually exclusive visible selections.
- FileList (p. 257) - Create a more sophisticated file-selection and browsing interface.
- Event Handling (p. 320) - Understand Gadget Factory's callback dispatcher.
- AddBinding (p. 329) - Understand native Vectric field registration.
- Dynamic HTML (p. 364) - Update HTML while retaining stable identifiers and bindings.
- AddStyle (p. 108) - Customize ListBox's appearance.
- InitializeScript (p. 34) - Construct and register components.

Summary

`GF.ListBox` creates a scrollable, single-selection control with programmer-defined labels and Lua values.

Its principal public methods are:

Method	Purpose
`New()`	Create a ListBox instance.
`WithContent()`	Prepare and install the component.
`Select()`	Select an item by index or clear the selection.
`Selected()`	Retrieve the selected value and index.
`SetItems()`	Replace the complete item collection.
`AddItem()`	Insert or append an item.
`RemoveItem()`	Remove an item.
`ReplaceItem()`	Replace an existing item.
`Clear()`	Remove all items.
`GetItems()`	Retrieve a copy of the item definitions.
`Count()`	Return the number of items.
`PrepareContent()`	Prepare the component without installation.
`GetHtml()`	Generate its HTML independently.

ListBox's appearance is produced through CSS, while JavaScript manages selection and keyboard navigation. One hidden Vectric-bound TextField carries the selected item identity back to Lua.

Its visible rows are ordinary HTML buttons, but their collective behavior presents a genuine list-selection interface.

ListBox demonstrates a central principle of Gadget Factory: the capabilities of an interface need not be limited to the native controls provided by its host.

REFERENCE 5.8

DataPicker

`GF.DataPicker` combines an editable text field with a dropdown of suggested strings. The user may choose a suggestion or type something else entirely.

The suggestions are conveniences, not a list of permitted values. This makes DataPicker useful for names, descriptions, and other text for which a few familiar choices save typing. Use Combobox (p. 192) when the result must identify one of its supplied items.

Loading the component

```lua
GF.LoadComponents {"DataPicker"}
```

The library loads its stylesheet automatically. Each instance has its own text, suggestions, and optional validity callback.

Basic usage

Create the picker during `InitializeScript()`:

```lua
local Suggestions = {"Roughing", "Finishing", "Engraving"}
local NamePicker

local function InitializeScript()
    GF.LoadComponents {"DataPicker"}
    NamePicker = GF.DataPicker.New("Operation name", {Text="Finishing"})
        :WithContent(Suggestions, 20, 60, 300)
end
```

Read `NamePicker:GetText()` when processing the results. It returns the text the user entered, whether or not it appears among the suggestions.

New

```lua
local picker = GF.DataPicker.New(title, options)
```

`title` supplies the heading and accessible label. A nil or empty title omits the visible heading. `options` is optional.

Option	Default	Meaning
`Text`	`""`	Initial text. Supply a string.
`Accent`	`"#4b5ba8"`	Accent color as `#RGB` or `#RRGGBB`.
`Disabled`	`false`	Initially disable text entry and suggestion selection.
`AllowBlanks`	`true`	Permit empty or whitespace-only text.
`OnValidityChanged`	nil	Optional function receiving the new Boolean validity.

Return value: The new DataPicker instance.

Disabling a picker does not make its text valid. Required text remains required.

WithContent

```lua
picker:WithContent(items, left, top, width)
```

`items` is a dense array of single-line strings. The other arguments specify pixel position and width; their defaults are `0`, `0`, and `200`.

Return value: The picker, allowing constructor and content calls to be chained.

The component copies the item array. An empty array is allowed and displays **No suggestions**. Empty strings, spaces, punctuation, and Unicode within suggestions are preserved. Text is escaped when incorporated into HTML; supply ordinary strings rather than pre-escaped HTML.

`WithContent()` installs the component's HTML, native binding, and handlers. Use Refresh() (p. 250) to replace suggestions after installation.

GetText

```lua
local text = picker:GetText()
```

While the dialog is bound, this reads the native text field. Before binding it returns the initial text; after Gadget Factory completes its cleanup it returns the cached final text.

The result is a string. No trimming, case conversion, lookup, or HTML escaping is applied to it.

Enabled

```lua
picker:Enabled(true)                        -- Enable editing and suggestions.
picker:Enabled(false)                       -- Disable them.
local enabled = picker:Enabled()            -- Read the current state.
picker:Enabled(not picker:Enabled())        -- Toggle it.
```

Return value: A Boolean representing the resulting enabled state. With no argument (or `nil`), the method only reads the state. Passing the current state again does nothing. As with `GF.ExecutionEnabled()`, a supplied value is interpreted using Lua's truth rules; normally pass `true` or `false`.

The initial state comes from `New()`'s `Disabled` option. `Enabled()` uses the positive form: `Disabled=true` means `Enabled()` returns `false`.

Disabling affects both the text field and the suggestion arrow, applies their disabled appearance, and closes any open suggestions. The entered text, suggestion contents, and native field binding are preserved. Reenabling leaves focus where it is; it does not reopen the list.

Call it before or after installation. Changes made before the dialog is ready, including during `InitializeDialog()`, are retained and applied by DataPicker's own post-load initialization. During interaction, the browser applies the new state through the component's internal update mechanism. The returned Boolean is the component's state; it is not a synchronous browser acknowledgment.

Changing the enabled state does not change `IsValid()` or generate an `OnValidityChanged` notification. If a disabled picker is irrelevant to an operation, the application should skip its validation, as The Layout Grid (p. 410) does for Guides.

Native Aspire acceptance - 23 October 2026: Petey confirmed that all requested `Enabled()` checks passed: deferred initialization, state queries, repeated setters and toggles, independent instances, retained text, and coordinated input/arrow availability. Disabling from a validity callback while the picker had focus closed its suggestions; typing, suggestion selection, and validity notifications continued after reenabling. State changes and queries did not generate validity notifications.

IsValid

```lua
local valid = picker:IsValid()
```

Return value: A Boolean. With the default `AllowBlanks=true`, any text is valid. With `AllowBlanks=false`, at least one character must match Lua's `%S` test for non-whitespace.

This is a blank-text rule, not a suggestion-membership rule. `"  Finishing  "` is valid and retains its spaces. `""` and `"  "` are invalid when blanks are disallowed.

`IsValid()` reads the current text without changing it, refreshing the suggestions, or invoking the callback. Call it again at the execution boundary when validity matters to the operation.

OnValidityChanged

```lua
local function ValidityChanged(valid)
    GF.ExecutionEnabled(valid)
end
```

Supply that function through `options.OnValidityChanged`. The callback receives **one Boolean argument**, with no instance argument. If several pickers share application logic, use a closure or an application-level refresh function to read the relevant instances.

The callback reports changes between valid and invalid:

Edit in a picker with `AllowBlanks=false`	Notification
Empty text becomes `"A"`	`true`
`"A"` becomes `"Another name"`	None: both are valid.
Valid text becomes empty or whitespace-only	`false`
One invalid blank value becomes another	None: both are invalid.

There is **no initialization notification**. Query initial validity in `InitializeDialog()`, once native fields have been bound:

```lua
local NameValid

local function InitializeDialog()
    NameValid = NamePicker:IsValid()
end

-- Include InitializeDialog = InitializeDialog in GF.GadgetDef.
```

Reading validity and changing the Execute button are separate operations. Native testing of InitializeDialog() (p. 49) found that DOM-dependent calls such as `GF.ExecutionEnabled()` cannot reliably update the button at that stage. Apply initial button readiness from an existing post-load callback (p. 367) or an explicit check action, as Example 6 does with **Check grid**. Do not invent a validity transition just to initialize the button.

If readiness also depends on other controls, have the validity callback call the application's full check function. A picker being valid does not establish that the whole operation is ready.

Ordinary typing, deletion, paste, and suggestion selection can produce a notification when they cross the validity boundary. Repeated browser events do not cause repeated Lua notifications for unchanged validity. The component preserves focus and the caret during these callbacks, while allowing normal Tab and mouse navigation.

The callback is optional. Omitting it leaves the editable picker usable without validity notifications. Its return value does not control dialog completion.

Refresh

```lua
picker:Refresh(items)
```

Return value: The picker.

`Refresh()` replaces the suggestions while retaining the text currently in the input. The text remains even when it is absent from the new suggestions. Refreshing closes an open suggestion list.

A subscribed picker also rechecks its text against the refreshed validation settings. For example, changing `picker.AllowBlanks` and refreshing can change the validity of an untouched field:

```lua
NamePicker.AllowBlanks = false
NamePicker:Refresh(Suggestions)
```

The browser checks the refreshed content and dispatches a callback only if validity changes. Do not depend on the callback having run before the Lua `Refresh()` call returns. Use `IsValid()` for a direct answer.

Refreshes before the native dialog is ready update the pending content. The initial state is still established without a synthetic notification.

Updating text through HTML_Dialog

DataPicker uses one permanent native text field. Its identifier is obtained through the usual instance method:

```lua
local textId = NamePicker:Id("text")
```

An external native update changes that field's value:

```lua
GF.Dialog:UpdateTextField(NamePicker:Id("text"), "Engraving")
NamePicker:Refresh(Suggestions)
```

The second line matters when the application needs a validity-change notification. **Native Aspire testing found that `UpdateTextField()` did not emit the tracked `input`, `keyup`, `change`, or value `propertychange` events.** DataPicker performs no recurring polling. `Refresh()` explicitly checks the new text and notifies if its validity changed.

Keep the suggestion array in your application so you can pass it back when refreshing. A refresh does not erase the newly assigned text.

`GetText()` and `IsValid()` can read the new value without a refresh; refreshing is needed to synchronize notification state, not to make the text readable. A direct JavaScript assignment likewise needs an explicit `input` or `change` event, or a component refresh, if a notification is required.

These are observed host behaviors, not a promise that every possible external writer emits the same events. The component's own suggestion selection already performs its explicit check.

Keyboard and popup behavior

The arrow button opens or closes the suggestions. Up and Down navigate them. Enter accepts the highlighted suggestion, and Escape closes an open list. Those popup keystrokes are consumed so they do not also complete or cancel the dialog. Tab closes the list and permits normal focus navigation.

The popup is positioned outside the component's clipping container and can open above the field when space below is limited. Refresh replaces the suggestion rows, not the native text input.

Preparing and rendering content independently

PrepareContent

```lua
picker:PrepareContent(items, left, top, width)
```

Prepares a new, uninstalled picker's content and layout without registering its native bindings. It returns the picker. Calling this after installation raises an error; use `Refresh()` instead.

GetHtml

```lua
local html = picker:GetHtml()
```

Returns the prepared component markup. Normal gadgets use `WithContent()`, which performs preparation, rendering, and registration together. Separate preparation and rendering are intended for tools that inspect component HTML.

Where to read more

- LayerPicker: layer-aware text entry (p. 252)
- Initializing native dialog state (p. 49)
- Controlling Execute readiness (p. 289)
- Native text updates and HTML replacement (p. 370)

REFERENCE 5.9

LayerPicker

`GF.LayerPicker` lets the user choose an existing layer name or type a new one. It supplies the current job's ordinary layer names as suggestions and uses DataPicker (p. 246) for editing and native field binding.

Choosing or typing a name does not create a layer, change the active layer, or add geometry. Those are separate decisions made by the gadget when it processes the results.

Loading the component

```lua
GF.LoadComponents {"LayerPicker"}
```

LayerPicker loads its DataPicker dependency and stylesheet automatically. It requires an open job when constructed; a typical gadget sets `Job=true` in `GF.GadgetDef`.

Basic usage

Create the picker during `InitializeScript()`:

```lua
local LP

local function InitializeScript(job)
    GF.LoadComponents {"LayerPicker"}
    LP = GF.LayerPicker.New("Output layer", {Text="My output"})
        :WithContent(20, 60, 300)
end
```

The field initially contains `"My output"`, even if that layer does not exist. Suggestions come from the job; they are not supplied to `WithContent()`.

New

```lua
local picker = GF.LayerPicker.New(title, options)
```

Option	Default	Meaning
`Text`	`""`	Initial layer name as a string.
`Accent`	`"#4b5ba8"`	Accent color as `#RGB` or `#RRGGBB`.
`Disabled`	`false`	Initially disable editing and suggestion selection.

Option	Default	Meaning
`OnValidityChanged`	nil	Optional function receiving the new Boolean validity.

Return value: The new LayerPicker instance.

LayerPicker always requires nonblank text. It does not expose an `AllowBlanks` option.

Construction enumerates the current job's layers. Layers marked `IsSystemLayer` are omitted from suggestions, and their names are recorded as prohibited names. The filter uses the API flag rather than a hard-coded spelling such as `ToolpathPreviews`.

WithContent

```lua
picker:WithContent(left, top, width)
```

Positions and installs the picker. The arguments are pixel coordinates and width, with defaults of `0`, `0`, and `200`.

Return value: The LayerPicker instance.

Unlike DataPicker, there is no `items` argument. LayerPicker owns its suggestions and native binding. Do not install its internal DataPicker separately.

GetText

```lua
local name = picker:GetText()
```

Returns the exact current text, preserving case and spaces. While bound, it reads the native field; after dialog cleanup it returns the cached final text.

This returns a name, not a `CadLayer` object. A valid name may refer to an existing ordinary layer or to a layer that has not yet been created.

Enabled

```lua
LP:Enabled(true)                    -- Enable the output-layer picker.
LP:Enabled(false)                   -- Disable it.
local enabled = LP:Enabled()        -- Read its current state.
LP:Enabled(not LP:Enabled())        -- Toggle it.
```

Return value: The resulting enabled state as a Boolean. Calling without an argument only reads it; requesting the current state again does nothing.

LayerPicker delegates to DataPicker's Enabled() method (p. 248). Both the text field and suggestion arrow change together, open suggestions close when disabled, and the entered name is retained. Changes before the dialog is ready are applied during the component's post-load initialization.

This does not enumerate, create, or activate layers, and it does not change name validity or generate a validity notification. The gadget decides whether an unavailable picker should participate in validation. See The Layout Grid (p. 410) for an example.

Native Aspire acceptance - 23 October 2026: Petey confirmed the delegated `Enabled()` behavior in the picker check gadget and the simplified Example 6 (p. 414). Initial state, queries, repeated toggles, retained names, and independent controls passed, as did mouse and keyboard output-mode changes with continued suggestion and validity behavior.

IsValid

```lua
local valid = picker:IsValid()
```

Return value: A Boolean. A name is valid when it contains at least one non-whitespace character and does not exactly match a current system-layer name.

The comparison is case sensitive. Names are not trimmed or rewritten. The system-layer check enumerates the current job, so a direct `IsValid()` call sees changes even before the displayed suggestions have been refreshed.

`IsValid()` is read-only: it does not create or activate layers, update the UI, or notify the consumer. Blank text returns false; checking a nonblank name requires an open job and raises an error if none exists. A gadget should not continue using the picker after its job context disappears.

The nonblank requirement is a **LayerPicker policy**. Native testing showed that Vectric can accept an empty layer name; the component deliberately requires a useful name for gadget operations.

Validity does not mean that every operation may write to the named layer. Ordinary bitmap, locked, or hidden layers are not excluded by LayerPicker's name validation. Apply the suitability checks required by your operation, as The Layout Grid (p. 411) does before inserting vectors.

Duplicate ordinary names are not resolved or renamed by LayerPicker. Vectric's duplicate-name lookup behavior remains unspecified here. If an ordinary layer shares a name with a system layer, that exact name is still rejected.

OnValidityChanged

```lua
local function Refresh()
    GF.ExecutionEnabled(LP:IsValid())
end

-- During InitializeScript, after loading LayerPicker:
LP = GF.LayerPicker.New("Output layer", {
    Text="My output", OnValidityChanged=Refresh
}):WithContent(20, 60, 300)
```

The callback receives one Boolean: `OnValidityChanged(valid)`. The example above ignores that argument because its refresh function can also serve an explicit check button and can be extended to check other controls.

There is no initialization notification. Query `IsValid()` in `InitializeDialog()` for the initial Boolean, but apply DOM-dependent button updates only after document loading. Native testing found that `GF.ExecutionEnabled()` cannot reliably update the button from `InitializeDialog()`; use an existing post-load callback or an explicit check action. See DataPicker's initialization guidance (p. 249). The optional callback reports later valid-to-invalid or invalid-to-valid transitions only.

Typing the first nonblank character, clearing the field, or completing an exact system-layer name can cross that boundary. Changing between two valid ordinary names does **not** notify, even if one is locked or hidden. A validity callback is not a general text-change callback.

When additional application rules matter, retain an explicit check action and revalidate at execution. Example 6 uses **Check grid** for that reason. It also repeats all layer-suitability checks before inserting anything.

DataPicker handles the event detection and preserves focus and selection during notification. There is no periodic timer and no Lua callback for every keystroke. The callback's Boolean comes from LayerPicker's authoritative current-job validation.

RefreshLayers

```lua
picker:RefreshLayers()
```

Re-enumerates the current job's layers, rebuilds the suggestions and system-name restrictions, and preserves the text currently being edited. It returns the LayerPicker instance.

Use it when the job's layers have changed while the dialog remains open. There is no automatic job-layer watcher. A subscribed picker can notify if the refreshed restrictions change the validity of its current text; an equivalent refresh does not duplicate the notification.

The browser processes the refreshed content. Do not rely on a callback having completed before `RefreshLayers()` returns; use `IsValid()` when an immediate Boolean is needed.

Updating text through HTML_Dialog

For an external native text update, obtain the field ID from the picker and then refresh:

```lua
GF.Dialog:UpdateTextField(LP:Id("text"), "Pocketing")
LP:RefreshLayers()
```

Native Aspire testing confirmed that `UpdateTextField()` did not produce the tracked editing or value `propertychange` events. The refresh supplies the explicit validity check that a callback requires. There is no polling fallback.

`GetText()` and `IsValid()` already read the updated text without a refresh. `RefreshLayers()` synchronizes the suggestions, system-name restrictions, and notification state; it does not erase the assigned name. A callback still occurs only if validity changes.

For the generic mechanism and direct JavaScript assignments, see DataPicker's native text updates (p. 250).

ActivateLayer

```lua
local action = picker:ActivateLayer()
local layer = action()
```

These two calls perform different work.

`ActivateLayer()` validates and captures the current name, then returns a reusable function. It does not create or activate a layer at that point. Later edits to the picker do not change the captured name.

Calling the returned function obtains the current job and checks its system-layer restrictions again. It finds the named layer or creates it if missing, rejects a system-layer result, activates the ordinary layer, and returns the `CadLayer` object. A missing job or prohibited name raises an error before intentional activation.

Use the action only after successful dialog completion, normally in `ProcessResults()` or in an execution plan run there. Calling it while the dialog is open would mutate the job during interaction.

The action captures a name, not a job or layer object. A gadget that must operate on the original job, sheet, or side should preserve and validate that context separately.

Direct insertion without activation

Some gadgets do not need to activate their destination layer. They can validate a name, find or create that layer after Execute, and insert objects directly with `layer:AddObject(...)`.

The Layout Grid (p. 411) takes this approach. It preserves the user's active layer and does not call `ActivateLayer()` merely to choose where its vectors belong.

Where to read more

- The underlying validity callback contract (p. 249)
- Requiring an open job (p. 11)
- Processing after Execute (p. 25)
- A complete LayerPicker example (p. 411)

REFERENCE 5.10

FileList

`GF.FileList` creates a scrollable file-selection control populated from the contents of a specified directory.

It allows the user to select a file matching a programmer-defined filter and provides an optional callback that receives information about the selected file.

Unlike ListBox (p. 217) and Combobox (p. 192), FileList does not support programmatic modification of its item collection during dialog execution. Its file list is established during initialization and remains fixed for that execution.

FileList also includes a built-in **details viewer and editor**, allowing descriptive information to be associated with individual files. This information is stored separately from the files themselves and is managed automatically by the component.

Although FileList contains considerably more functionality than an ordinary selection control, its public programming interface is deliberately small.

Loading the component

FileList is supplied as a separate library module and must be loaded before use.

```lua
GF.LoadComponents {"FileList"}
```

This exposes the `GF.FileList` constructor and includes its companion stylesheet, `FileList.css`.

The library is loaded once per gadget. Multiple independent FileList instances may then be created.

Basic usage

A FileList is created using `New()` and populated using `WithContent()`.

```lua
local Templates = GF.FileList.New("Surfacing Templates")
    :WithContent(
        [[D:\CNC\Templates]],
        "*.ToolpathTemplate",
        30, 50, 350, 280
    )
```

This creates a FileList titled **Surfacing Templates**, positioned 30 pixels from the left and 50 pixels from the top.

The component occupies an area 350 pixels wide and 280 pixels high.

It displays the files in the specified directory whose names match the `*.ToolpathTemplate` filter.

FileList automatically enumerates the matching files, sorts them alphabetically, and constructs the HTML required to display them.

The user can select a file by clicking its entry.

Construction sequence

The normal construction sequence consists of two operations:

1. `New()` creates the FileList instance and establishes its optional selection callback.
2. `WithContent()` reads the directory, prepares the file collection, generates the interface, and registers the required bindings and event handlers.

Both methods return the FileList instance, supporting the fluent construction style shown above.

New

`FileList.New()` creates a new FileList instance.

Syntax

```lua
local list = GF.FileList.New(title, onselect)
```

Parameters

Parameter	Description
`title`	Optional heading displayed above the file list.
`onselect`	Optional Lua function invoked when the user selects a different file.

Return value: A new FileList instance.

Without a callback

```lua
local Templates = GF.FileList.New("Templates")
```

The component operates normally without a programmer-defined callback.

The user can select files and examine their associated details. The selected file can be retrieved later using `Selected()`.

This is appropriate when the file is needed only after the user finishes interacting with the dialog.

With a callback

```lua
local TemplateChanged = function(file, list)
    SelectedTemplate = file.path
end

local Templates = GF.FileList.New(
    "Surfacing Templates",
    TemplateChanged
)
```

The callback receives two arguments:

Argument	Description
`file`	Table containing information about the selected file.
`list`	FileList instance that generated the event.

The originating instance can be used when several FileLists share one callback.

Optional title

If `title` is omitted, `nil`, or an empty string, the component does not generate a heading.

The list and its details facilities remain available.

WithContent

`WithContent()` identifies the directory to examine and establishes the component's position and dimensions.

Syntax

```lua
list:WithContent(folder, mask, left, top, width, height)
```

Parameters

Parameter	Description
`folder`	Directory containing the files to display.
`mask`	Filename filter used when enumerating files.
`left`	Horizontal position in pixels.
`top`	Vertical position in pixels.
`width`	Width of the component in pixels.
`height`	Height of the component in pixels.

Return value: The FileList instance.

Directory selection

The `folder` argument specifies the directory to search.

```lua
local Folder = [[D:\CNC\Templates]]
```

FileList automatically appends a trailing backslash when one is not already present.

The directory should exist and be accessible to the current user.

Unlike a file-picker dialog, FileList does not ask the user to choose a directory at runtime. The programmer establishes the directory during initialization.

Filename filtering

The `mask` parameter is passed to Vectric's native `DirectoryReader`.

Typical examples include:

```lua
"*.ToolpathTemplate"
"*.txt"
"*.csv"
"*.*"
```

For example:

```lua
Templates:WithContent(
    Folder,
    "*.ToolpathTemplate",
    30, 50, 350, 280
)
```

displays only files matching the specified template extension.

Directory enumeration

Internally, FileList uses Vectric's `DirectoryReader`:

```lua
local dr = DirectoryReader()

dr:BuildDirectoryList(self.path, false)
dr:GetFiles(mask, true, false)
dr:SortFiles(DirectoryReader.SORT_ALPHA, false)
```

These operations:

1. Establish the directory to search.
2. Collect matching files.
3. Sort the results alphabetically.

The `false` argument to `BuildDirectoryList()` prevents recursive traversal into subdirectories.

FileList therefore examines the specified directory rather than its entire directory tree.

Directory enumeration takes place when `WithContent()` prepares the component, before the dialog is displayed.

Position and dimensions

The remaining arguments establish the component's outer bounds.

```lua
Templates:WithContent(Folder, "*.txt", 20, 60, 300, 250)
```

The generated HTML uses these values as pixel dimensions.

FileList's current implementation inserts them through Lua's integer formatting (`%d`), so whole-number coordinates and dimensions should be supplied.

Unlike some newer components, FileList does not provide default geometry values or explicit parameter validation.

Static file collection

The file list is established during initialization.

Files subsequently created or removed from the directory are not automatically reflected in the displayed collection.

FileList does not expose `SetItems()`, `AddItem()`, or other runtime collection-management methods.

For an interface requiring a dynamically changing collection, consider ListBox (p. 217) or Combobox (p. 192).

Registration

`WithContent()` performs the following operations:

1. Reads the matching files using `DirectoryReader`.
2. Loads any existing details information.
3. Prepares the HTML for the file list, details viewer, and editor.
4. Registers handlers for the individual file buttons.
5. Registers handlers for the editor's controls.
6. Registers the component with Gadget Factory for native field binding.

The method should normally be called once per instance during InitializeScript() (p. 34).

Repeated calls to `WithContent()` are not a supported way to refresh the list and may produce duplicate handler registrations.

Selected

`Selected()` retrieves information about the currently selected file.

Syntax

```lua
local file = list:Selected()
```

Return value

A table containing the selected file's information, or `nil` if no file is selected.

The returned table contains two fields:

Field	Description
`path`	Full pathname of the selected file.
`name`	Filename without its extension.

Example

```lua
local file = Templates:Selected()

if file then
    SelectedPath = file.path
    SelectedName = file.name
end
```

For a selected file named `MDF_Surface.ToolpathTemplate`, the returned information is equivalent to:

```lua
{
    path = [[D:\CNC\Templates\MDF_Surface.ToolpathTemplate]],
    name = "MDF_Surface"
}
```

The actual path and name originate from Vectric's `FileEntry.Name` and `FileEntry.FileNameNoExt` properties.

No selection

FileList does not establish a default selection.

Until the user selects a file:

```lua
Templates:Selected()  -- nil
```

Programmers should check for this condition before using the returned value.

For example:

```lua
local file = Templates:Selected()

if not file then
    return
end

-- Use file.path
```

Selection semantics

Unlike RadioGroup (p. 120), Combobox (p. 192), and ListBox (p. 217), FileList's `Selected()` method does not return a separate selection index.

It returns the selected file-information table directly.

FileList also does not provide a public `Select(index)` method.

The component's normal selection mechanism is user interaction with the displayed file entries.

Returned table ownership

The returned table is the corresponding entry in FileList's internal file collection, not a newly created copy.

It should be treated as read-only application information.

Modifying its fields may interfere with the component's own operations.

Selection callbacks

An optional callback allows the gadget to respond immediately when the user selects a file.

Example

```lua
local TemplateSelected = function(file, list)
    CurrentTemplate = file.path
    GF.ExecutionEnabled(true)
end
```

Create the component with that callback:

```lua
local Templates = GF.FileList.New(
    "Surfacing Templates",
    TemplateSelected
):WithContent(
    [[D:\CNC\Templates]],
    "*.ToolpathTemplate",
    30, 50, 350, 280
)
```

When the user selects a file, FileList invokes the callback with the selected file table and its own instance.

The example also enables Gadget Factory's standard Execute button, which is initially inactive.

This is a common FileList pattern: **require the user to select a valid file before allowing the gadget to proceed**.

Callback arguments

A callback receives:

```lua
onselect(file, list)
```

The `file` argument contains the same table returned by `Selected()`.

The `list` argument identifies the originating FileList instance.

Shared callbacks

Several FileLists can use a common callback.

```lua
local FileSelected = function(file, list)
    if list == TemplateList then
        SelectedTemplate = file.path
    elseif list == ProfileList then
        SelectedProfile = file.path
    end
end
```

This follows the same general callback convention used by Gadget Factory's other components.

Repeated selection

FileList does not invoke the callback when the user clicks an item that is already selected.

Its selection handler begins by checking whether the clicked index matches the current selection.

If it does, no further processing occurs.

Selection after dialog completion

FileList retains the selected file in its instance state.

Consequently, the selection remains available during the gadget's normal `ProcessResults()` callback.

For example:

```lua
local function ProcessResults()
    local file = Templates:Selected()

    if not file then return end

    -- Process the selected file
    -- using file.path
end
```

This is useful when the gadget needs to load or process the chosen file after the dialog has closed.

Example: Selecting a surfacing template

The following abbreviated example illustrates the normal FileList workflow.

```lua
local Templates

local TemplateSelected = function(file, list)
    GF.ExecutionEnabled(file ~= nil)
end

local function InitializeScript(job)
    GF.LoadComponents {"FileList"}

    Templates = GF.FileList.New(
```

LUA / CONTINUED

```lua
        "Surfacing Templates",
        TemplateSelected
    ):WithContent(
        [[D:\CNC\Templates]],
        "*.ToolpathTemplate",
        30, 50, 350, 280
    )
end

local function ProcessResults()
    local selected = Templates:Selected()

    if not selected then return end

    -- Load or process selected.path
end
```

The FileList is created during initialization.

When the user selects a file, the callback enables the standard Execute button.

After successful dialog completion, `ProcessResults()` retrieves the selected file.

This example is intentionally limited to FileList usage. The actual processing of Vectric toolpath templates is a separate application responsibility.

The built-in details viewer

FileList includes a details area beneath its file-selection list.

This area displays descriptive information associated with the selected file.

Unlike a conventional list control, FileList can therefore present both the available files and explanatory information about them within one component.

The details viewer is built into FileList. It does not require additional configuration or a separate component.

Initial information

When no file is selected, the details area displays general comments associated with the directory, when such comments are available.

When a file is selected, FileList displays its corresponding details.

If that file has no stored details, the area is empty.

Editing details

A small Edit button opens the built-in details editor.

The editor provides a text area and controls for saving or canceling the current edit.

The editor can modify:

- General comments associated with the directory.

- Descriptive information associated with an individual file.

These facilities are managed internally by FileList.

The gadget programmer does not need to create the editor's HTML, register its controls, or manage its callbacks.

Details storage

FileList stores descriptions in a file named:

```lua
Details.ini
```

This file is located in the directory supplied to `WithContent()`.

Individual file descriptions are organized into sections identified by the filenames without their extensions.

For example:

```text
Templates for surfacing operations.

[MDF_Surface]
Use for normal MDF spoilboard surfacing.

[Finish_Pass]
Use for a light finishing pass.
```

The component reads this file during content preparation and associates its sections with the matching files.

General comments appear before the file-specific sections.

The details editor does not modify the selected files themselves. It maintains separate descriptive metadata in `Details.ini`.

When details are saved

Clicking the editor's Save control updates FileList's in-memory details collection.

Modified details are written to `Details.ini` during Gadget Factory's normal component cleanup, when the dialog finishes.

This distinction matters because saving a description in the editor does not immediately guarantee that it has been written to disk.

The current implementation also does not distinguish between successful gadget execution and normal dialog cancellation when saving committed details edits.

Consequently, **choosing the gadget's Cancel button does not automatically discard description changes previously saved in FileList's editor**.

The editor's own Cancel button, by contrast, abandons the current uncommitted edit.

Persistence considerations

The details file is managed automatically, but several behaviors are worth knowing:

- FileList writes descriptions only when changes have been made.
- Only descriptions associated with files still present in the prepared file collection are written.
- When no descriptions or general comments remain, an existing `Details.ini` may be deleted.
- File descriptions are identified by names without extensions.
- Two files with the same base name but different extensions can therefore share a details identifier.
- The directory must be writable for edited descriptions to be persisted successfully.

Because `Details.ini` is a fixed filename, a directory used with FileList should not contain an unrelated file of the same name that must be preserved.

For ordinary gadget development, these details are largely transparent. The component handles their loading, editing, and persistence without additional programmer code.

How FileList works internally

FileList predates several of Gadget Factory's newer interface components.

Its implementation demonstrates an earlier - but still effective - approach to creating sophisticated interfaces using HTML controls and Vectric callbacks.

The basic architecture consists of:

1. Vectric's `DirectoryReader` for enumerating files.
2. HTML buttons representing the individual file entries.
3. Gadget Factory's centralized event dispatcher.
4. Native Vectric LabelFields and a TextField for the details interface.
5. CSS and dynamic HTML for selection highlighting and editing modes.

Automatic identifiers

Each FileList instance receives an automatically generated identifier.

The first instance is named:

```lua
fl1
```

Its internal controls receive identifiers resembling:

```lua
lstFiles_fl1
labDetails_fl1
txtDetails_fl1
labEditor_fl1
btnEdit_fl1
btnEditSave_fl1
btnEditCancel_fl1
```

Individual file entries use identifiers such as:

```lua
fli1_fl1
fli2_fl1
fli3_fl1
```

These identifiers are generated automatically and are normally of no concern to the gadget programmer.

Buttons disguised as list entries

Each file is represented by an ordinary HTML button using Vectric's special `LuaButton` class.

For example:

```html
<input type="button"
       id="fli1_fl1"
       class="LuaButton"
       value="MDF_Surface">
```

The component's stylesheet removes the usual button appearance and makes each button resemble a conventional list item.

The file's extension is omitted from the displayed label.

Unlike ListBox (p. 217), which delegates row events through one shared handler, FileList registers a separate Gadget Factory handler for each file entry.

Because the file collection is static, those handlers need only be established once.

File-selection handling

When a file entry is activated, FileList performs several operations:

1. Ignore the event if the item is already selected.
2. Remove the previous item's selection highlighting.
3. Highlight the newly selected item.
4. Display the new file's associated details.
5. Store the selected item index.
6. Invoke the programmer's optional `onselect` callback.

The component changes the selected appearance using Vectric's `SetOuterHtml()` method.

The corresponding HTML button is replaced with another button using the same ID and an inline style that indicates selection.

This takes advantage of the dynamic HTML behavior discussed in Dynamic HTML (p. 364): the element's identity remains recognizable to Gadget Factory's existing callback dispatcher.

Native field bindings

FileList does not register its file-selection buttons as ordinary Vectric data fields.

They generate LuaButton events directly.

However, the details interface uses native Vectric field bindings.

The component's `Bind()` method contains:

```lua
function FileList:Bind(dlg)
    dlg:AddLabelField(self.labEditor, "")
    dlg:AddLabelField(
        self.labDetails,
        self.details[Comments] or ""
    )
    dlg:AddTextField(self.txtDetails, "")
    self.dlg = dlg
end
```

The two LabelFields provide the details display and editor heading.

The TextField supports the editor's text entry.

The component retains the native dialog reference for subsequent updates.

This is a useful example of combining ordinary HTML buttons with native-bound display and editing fields inside a single component.

Component cleanup

FileList implements a `Done()` method that Gadget Factory invokes during its normal cleanup sequence.

If descriptions were modified, this method writes the current details collection to `Details.ini`.

The component therefore manages both its interface and the persistence of its descriptive information.

Programmers do not normally call `Bind()` or `Done()` directly.

Preparing and rendering content independently

Like the newer Gadget Factory components, FileList exposes separate methods for content preparation and HTML generation.

PrepareContent

```lua
list:PrepareContent(folder, mask, left, top, width, height)
```

Accepts the same arguments as `WithContent()`.

Return value: The FileList instance.

The method reads the directory, loads the details information, resets the current selection and editing state, and prepares the file collection.

It does not register Gadget Factory event handlers or native Vectric bindings.

Unlike a purely computational preparation function, `PrepareContent()` performs real filesystem operations.

It should therefore not be treated as a harmless rendering refresh on an already installed instance.

GetHtml

```lua
local html = list:GetHtml()
```

Return value: A string containing the complete FileList interface.

The generated markup includes:

- The outer FileList container.
- The optional heading.
- The scrollable file list.
- The details display.
- The details editor.
- The editor's action buttons.

`GetHtml()` does not register event handlers or native field bindings.

Example

```lua
local Preview = GF.FileList.New("Templates")

Preview:PrepareContent(
    [[D:\CNC\Templates]],
    "*.ToolpathTemplate",
    20, 50, 300, 250
)

local html = Preview:GetHtml()
```

This produces FileList's HTML representation without installing it as an active Gadget Factory component.

The directory is still read during `PrepareContent()`.

The returned HTML alone does not establish a working file-selection and details-editing interface.

For normal gadget development, use `WithContent()`.

Relationship to WithContent

The normal installation sequence is:

```lua
function FileList:WithContent(
    folder, mask, left, top, width, height
)
    self:PrepareContent(
        folder, mask, left, top, width, height
    )

    -- Register file and editor handlers
    -- Register the component and its HTML

    return self
end
```

As with the other components, the distinction is:

PrepareContent gathers and prepares the data. GetHtml renders the interface. WithContent installs the working component.

Styling and layout

The accompanying `FileList.css` stylesheet controls the component's layout and appearance.

Main container

FileList's outer container is absolutely positioned using the coordinates and dimensions supplied to `WithContent()`.

The optional title appears above the main list area.

Because the title is positioned with a negative top offset, the programmer should allow space for it when placing the component near the top of the dialog.

File list and details area

The component divides its height between two regions:

Region	Approximate share
Scrollable file list	65%
Details area	Remaining 35%

The details region has a small separation from the file list.

This division is established through CSS and does not require the programmer to provide separate dimensions.

File entry appearance

The stylesheet uses rules such as:

```css
.FileList input[type="button"][id^="fli"] {
    display: block;
    width: 100%;
    height: 28px;
    margin: 0;
    padding: 2px 6px;
    background-color: white;
    border: 0;
    text-align: left;
    font-weight: normal;
}
```

These rules make the individual file buttons resemble list rows.

A selected item receives a light-blue background and border through dynamically generated inline styling.

Details editor

The editor occupies an overlay within the FileList container.

Its visibility is controlled using the `editing` CSS class on the outer container.

For example:

```css
.FileList.editing .FileList-editor {
    visibility: visible;
}
```

Related rules show the editor's Save and Cancel buttons while hiding the ordinary Edit button.

This allows FileList to change between viewing and editing modes without creating a separate dialog.

Gadget-specific styling

The programmer may override the component's appearance through AddStyle (p. 108).

For example:

```lua
GF.AddStyle([[
    .FileList-list {
        background-color: #f8fbff;
    }

    .FileList input[type="button"][id^="fli"] {
```

LUA / CONTINUED

```
        height: 32px;
    }
]])
```

These changes affect presentation without modifying the FileList module.

As with other Gadget Factory components, Vectric's embedded browser determines which CSS features are available.

FileList versus the other list controls

FileList, ListBox, and Combobox share the general purpose of selecting an item, but differ in how their contents are obtained and managed.

Feature	FileList	ListBox	Combobox
Item source	Filesystem directory	Lua item table	Lua item table
Presentation	Scrollable file list	Scrollable list	Dropdown
Item value	File-information table	Programmer-defined	Programmer-defined
Initial selection	None	None	None
Runtime item modification	Not supported	Supported	Supported
Selection callback	`(file, list)`	`(value, list)`	`(value, combo)`
Programmatic selection	Not exposed	`Select()`	`Select()`
Details viewer/editor	Built in	Not provided	Not provided
Metadata persistence	`Details.ini`	Not provided	Not provided

The appropriate component depends on the application's purpose.

Use Combobox (p. 192) when a compact, dynamically managed selection is needed.

Use ListBox (p. 217) when the choices should remain visible and may change during execution.

Use FileList when the choices are files in a known directory and the user may benefit from accompanying descriptive information.

Important considerations

- **Library loading:** Load FileList through `GF.LoadComponents()` before constructing an instance.
- **Multiple instances:** Each instance receives automatically generated identifiers.

- **Directory requirement:** The supplied folder must be suitable for Vectric's `DirectoryReader`.
- **Nonrecursive search:** FileList examines the specified directory without traversing its subdirectories.
- **Filename filtering:** The mask is passed directly to `DirectoryReader`.
- **Alphabetical order:** Matching files are sorted before generating the list.
- **Static collection:** FileList does not automatically refresh when directory contents change.
- **No initial selection:** `Selected()` returns `nil` until the user selects an entry.
- **Selection return type:** `Selected()` returns a file-information table, not a string or selection index.
- **No public selection setter:** FileList does not expose a `Select()` method.
- **Callback arguments:** The selection callback receives the selected file table and originating instance.
- **Repeated selections:** Clicking the already selected item does not invoke the callback again.
- **Native fields:** The details interface uses native LabelField and TextField bindings; file selection itself uses LuaButton events.
- **Built-in metadata:** Descriptions are stored separately from the selected files in `Details.ini`.
- **Editor persistence:** Saved edits are normally written to disk during component cleanup, regardless of whether the gadget was completed or canceled.
- **Writable directory:** The details editor requires write access when saving modifications.
- **Base-name identification:** Details are keyed by filenames without extensions, which may cause collisions when different files share a base name.
- **No automatic HTML escaping:** Filenames, titles, and details should originate from trusted sources; the component does not consistently escape them for HTML.
- **One-time installation:** `WithContent()` is intended to be called once per component instance.
- **Read-only HTML generation:** `GetHtml()` generates markup without installing the component, while `PrepareContent()` still reads the filesystem.
- **Dialog lifecycle:** FileList is normally created during InitializeScript() (p. 34) and bound when Gadget Factory creates the native dialog.

Related functions

- LoadComponents (p. 36) - Load Gadget Factory component libraries.
- ListBox (p. 217) - Create a dynamically managed, scrollable list.
- Combobox (p. 192) - Create a dynamically managed dropdown.
- Event Handling (p. 320) - Understand Gadget Factory's LuaButton dispatcher.
- AddBinding (p. 329) - Understand native Vectric field registration.
- Dynamic HTML (p. 364) - Understand HTML replacement and preserved control identifiers.
- Default Buttons (p. 288) - Enable Execute after a valid file has been selected.
- AddStyle (p. 108) - Customize the component's appearance.

- InitializeScript (p. 34) - Construct and register FileList.

Summary

`GF.FileList` provides a file-selection interface with integrated descriptive metadata.

Its principal public methods are:

Method	Purpose
`New()`	Create a FileList instance.
`WithContent()`	Enumerate files and install the working interface.
`Selected()`	Retrieve the selected file-information table.
`PrepareContent()`	Read and prepare the file collection without registration.
`GetHtml()`	Generate the component's HTML independently.

The component uses Vectric's `DirectoryReader` to collect files, HTML buttons to display them, Gadget Factory's callback dispatcher to handle selection, and native fields to support its details editor.

Its filesystem-backed collection and integrated metadata editor distinguish it from the other list components.

FileList does more than ask which file the user wants. It provides enough context to help them make that choice.

CHAPTER 06

06

Utilities

Formatting, completion controls, and useful feedback.

IN THIS CHAPTER

Utilities: overview 278

NumberFormatter 281

Default Buttons 288

StatusBar 298

REFERENCE 6.1

Utilities: overview

Gadget Factory provides several utility functions that simplify common programming tasks without requiring the programmer to construct additional HTML controls or develop specialized components.

These facilities address three recurring needs: **numeric formatting, dialog completion, and user feedback**.

Unlike the framework's HTML primitives, which concentrate on constructing interface elements, the utilities provide convenient solutions to tasks that arise throughout gadget development.

Each utility can be used independently, and none requires the programmer to understand its internal implementation.

Available utilities

Utility	Purpose
NumberFormatter (p. 281)	Create reusable functions that format numeric values to a specified precision.
Default Buttons (p. 288)	Manage the standard Execute and Cancel buttons and their dialog-completion behavior.
StatusBar (p. 298)	Display information, progress, and visual status indications without interrupting the user.

NumberFormatter

`GF.NumberFormatter()` generates a reusable function that formats numbers to a specified number of decimal places, removing unnecessary trailing zeros.

For example:

```lua
local D3 = GF.NumberFormatter(3)

local value = D3(12.34567)  -- "12.346"
```

The generated formatter remembers its precision, eliminating the need to specify that precision for every value.

The NumberFormatter (p. 281) chapter explains this *function factory* technique, including Lua closures and the distinction between numeric values and formatted strings.

Default Buttons

Gadget Factory supplies standard Execute and Cancel controls that integrate with Vectric's native dialog completion mechanism.

These functions support two different button-management approaches.

Call `GF.NoDefaultButtons()` during InitializeScript() (p. 34) to suppress both standard buttons when the gadget supplies its own completion controls:

```lua
-- Inside InitializeScript()
GF.NoDefaultButtons()
```

When using the standard buttons, call `GF.ExecutionEnabled(true)` to enable Execute after the dialog's HTML document is ready. Make this call from a callback after the document has loaded:

```lua
-- In a callback after the HTML document has loaded
GF.ExecutionEnabled(true)
```

These examples represent different approaches, not consecutive initialization operations.

`GF.ExecutionEnabled(false)` disables Execute; `GF.ExecutionEnabled()` reports the current execution-enabled state.

See Default Buttons (p. 288) for their usage, initialization requirements, and relationship to `ProcessResults()` and `DiscardResults()`.

StatusBar

`GF.StatusBar` provides a persistent display along the bottom of a gadget dialog.

It supports independently configured left and right text areas, optional formatting functions, and three visual states: Normal, Caution, and Warning.

Unlike the other utilities in this section, StatusBar is a singleton supplied through a separately loaded library module.

```lua
GF.LoadComponents {"StatusBar"}

GF.StatusBar.Init {
    GetLeft = "Ready"
}
```

Once the dialog is running, the displayed information and visual state can be updated programmatically.

```lua
GF.StatusBar.Warning(true)
```

The StatusBar is particularly useful for routine validation messages and operational feedback that would otherwise require disruptive modal message boxes.

See StatusBar (p. 298) for configuration, getter functions, text updates, visual states, and formatting options.

Initialization and runtime use

Although grouped together for convenience, these utilities participate in different stages of the gadget lifecycle.

Utility	Typical usage
NumberFormatter	Create formatting functions during script setup; use them wherever formatting is needed.
NoDefaultButtons	Configure the dialog during InitializeScript() (p. 34).
ExecutionEnabled	Change execution availability after the dialog's HTML document is ready.
StatusBar.Init	Configure and register the StatusBar during initialization.
StatusBar.Update and state methods	Refresh information or change status during dialog execution.

Understanding this distinction helps avoid attempting runtime operations before the native dialog exists.

The InitializeScript (p. 34) and InitializeDialog (p. 49) chapters explain the broader lifecycle.

Choosing the appropriate facility

Utilities are intended to eliminate repetitive work, not impose a particular programming style.

A simple gadget may need only the default buttons. Another may require consistent numeric formatting but no StatusBar. A more interactive gadget may use all three.

The programmer remains free to construct custom alternatives when the supplied facilities do not meet the application's needs.

For more specialized behavior, see Advanced Topics (p. 316), which explains the underlying event, binding, and HTML mechanisms available for extending Gadget Factory.

Summary

Gadget Factory's utilities provide convenient, reusable solutions to tasks that are common across otherwise unrelated gadgets.

They reduce boilerplate while preserving the programmer's control over the application's behavior and presentation.

Use the utilities when they meet the requirement. Build something different when they don't.

That balance between convenience and flexibility is central to Gadget Factory's design.

REFERENCE 6.2

NumberFormatter

`GF.NumberFormatter()` creates a reusable function that converts numbers into consistently formatted strings.

The generated function rounds values to a specified number of decimal places, removes unnecessary trailing zeros, and eliminates negative zero.

Unlike most Gadget Factory functions, `NumberFormatter()` **does not perform its primary operation directly. Instead, it returns another function that performs the formatting.**

This programming technique is known as a *function factory*.

Syntax

```lua
local D = GF.NumberFormatter(places)
local result = D(value)
```

Parameters

Parameter	Description
`places`	Number of decimal places to retain. Use a positive integer.
`value`	The numeric value passed to the generated formatting function.

Return values

- `GF.NumberFormatter(places)` returns a **function**.
- `D(value)` returns a **string** containing the formatted number.

The identifier `D` is simply a variable name chosen by the programmer. Any valid Lua identifier can be used.

Basic example

```lua
local D = GF.NumberFormatter(3)

local X = 12.34567
local Y = 8.50000

local formattedX = D(X)  -- "12.346"
local formattedY = D(Y)  -- "8.5"
```

The formatter rounds each number to a maximum of three decimal places, without displaying insignificant trailing zeros.

Formatting results

Given `local D = GF.NumberFormatter(3)`:

Input	Result
`12.34567`	`"12.346"`
`12.3`	`"12.3"`
`12.0`	`"12"`
`100.0`	`"100"`
`0.0004`	`"0"`
`-0.0004`	`"0"`

Notice that every result is a string, even when its contents resemble an integer.

Understanding function factories

In traditional procedural programming, a function normally performs an operation and returns a result.

For example:

```lua
local square = function(n)
    return n * n
end

local result = square(5)  -- 25
```

`NumberFormatter()` behaves differently. Its job is to **create a new function configured for a particular purpose**.

Consider:

```lua
local D2 = GF.NumberFormatter(2)
local D4 = GF.NumberFormatter(4)
```

We now have two independently configured formatting functions.

```lua
local value = 12.34567

D2(value)  -- "12.35"
D4(value)  -- "12.3457"
```

The original `NumberFormatter()` function was called twice, creating two formatters with different precision settings.

Each generated function remembers its own configuration.

How does a function remember?

Here is the actual Gadget Factory implementation:

```lua
local function NumberFormatter(places)
    local fmt = "%." .. places .. "f"

    return function(value)
        local s = string.format(fmt, value):gsub("%.?0+$", "")
        return s == "-0" and "0" or s
    end
end
```

Let's examine the sequence.

1. The outer function receives the configuration.

```lua
local D = GF.NumberFormatter(3)
```

Inside `NumberFormatter()`, the first statement constructs the format string:

```lua
local fmt = "%.3f"
```

This tells Lua's `string.format()` to produce a number with three digits after the decimal point.

2. The outer function returns an inner function.

```lua
return function(value)
    ...
end
```

The inner function is not executed at this point. It is returned as a value and assigned to `D`.

3. The inner function remembers `fmt` .

Ordinarily, local variables cease to be accessible when their function finishes executing.

Lua provides an important exception: when an inner function references a local variable belonging to an enclosing function, that variable remains accessible for as long as the inner function needs it.

This mechanism is called a **closure**.

The remembered variable is called an *upvalue*.

In our example, the returned function retains access to `fmt`, even though `NumberFormatter()` has already returned.

4. Calling `D()` executes the generated function.

```lua
local result = D(12.34567)
```

The inner function uses its remembered format string to process the supplied value.

This distinction is fundamental:

```lua
GF.NumberFormatter(3)  -- Creates and returns a function
D(12.34567)            -- Executes that function
```

A function factory is therefore a way of configuring an operation once and reusing it repeatedly without resupplying the configuration.

How the formatting works

The generated function performs three operations.

1. Round to the requested precision

```lua
string.format(fmt, value)
```

Given `fmt = "%.3f"` and `value = 12.34567`, the result is:

```text
12.346
```

Values such as `12.3` initially become `"12.300"`.

2. Remove unnecessary trailing zeros

```lua
:gsub("%.?0+$", "")
```

The Lua pattern removes trailing zeros and, when appropriate, the decimal point preceding them.

Examples:

```text
"12.300" → "12.3"
"12.000" → "12"
"12.346" → "12.346"
```

3. Normalize negative zero

Very small negative numbers can round to zero while retaining their negative sign.

The final statement addresses this:

```lua
return s == "-0" and "0" or s
```

If the formatted result is `"-0"`, the function returns `"0"` instead.

Otherwise, it returns the formatted string unchanged.

Example: Sectio Divina

Sectio Divina creates its formatter immediately after loading Gadget Factory:

```lua
local GF = assert(loadfile(root .. "GadgetFactory.gfm"), "Cannot load GadgetFactory.")(root)

local D = GF.NumberFormatter(3)
```

This makes `D()` available throughout the gadget.

Formatting initial label values

The dimensions display uses the formatter inside an AddElements() (p. 74) collection:

```lua
{GF.AddLabelSpan, "labX", D(X)}, " × ", {GF.AddLabelSpan, "labY", D(Y)},
```

Both numeric values are converted to formatted strings before they are passed to `GF.AddLabelSpan()`.

The multiplication symbol is inserted separately as ordinary HTML text.

Updating labels

Sectio Divina also uses the formatter when dimensions change:

```lua
Dlg:UpdateLabelField("labX", D(X))
Dlg:UpdateLabelField("labY", D(Y))
```

This ensures that the displayed dimensions use the same precision and trailing-zero rules as the initial values.

Updating numeric fields

Numeric fields require a slightly different treatment:

```lua
local SetDoubleFields = function(x, y)
    if x then Dlg:UpdateDoubleField("txtX", tonumber(D(x))) end
    if y then Dlg:UpdateDoubleField("txtY", tonumber(D(y))) end
end
```

Here, `tonumber()` converts the formatted string back into a Lua number.

This distinction matters because `UpdateDoubleField()` expects a numeric value rather than display text.

The conversion also means the value supplied to the field has been rounded to the formatter's precision.

Important considerations

Formatting does not change the original number

```lua
local X = 12.34567
local text = D(X)

-- X remains 12.34567
-- text contains "12.346"
```

`NumberFormatter()` is intended primarily for display formatting.

Use `tonumber(D(value))` only when the rounded numeric result is specifically desired.

The precision is a maximum, not a fixed display width

Although `string.format()` initially generates the specified number of decimal places, the formatter removes unnecessary zeros.

Therefore, `GF.NumberFormatter(3)` does not guarantee that three decimal digits will appear in the final string.

Zero decimal places are not supported correctly

The current implementation has a limitation when `places` is zero.

For example:

```lua
local D0 = GF.NumberFormatter(0)

D0(100)  -- "1"
D0(0)    -- ""
```

Because the trailing-zero removal pattern also matches zeros in integers without decimal points, significant digits may be removed.

Until the implementation is revised, use a positive integer for `places`.

Invalid arguments

The function does not validate its arguments.

Omitting `places`, supplying an invalid precision, or formatting a value that `string.format()` cannot interpret numerically may cause a Lua error.

Summary

`GF.NumberFormatter()` is a function factory that creates reusable numeric formatting functions.

Its principal characteristics are:

- Precision is configured when the formatter is created.
- The returned function remembers that configuration through a Lua closure.
- Values are rounded to the configured number of decimal places.
- Unnecessary trailing zeros and decimal points are removed.
- Negative zero is normalized.
- Formatted results are returned as strings.

The most important distinction is that `NumberFormatter()` **creates the formatting function; the returned function formats the number.**

See also

- AddElements() (p. 74)
- AddLabelSpan() (p. 103)
- AddDoubleField (p. 97)

REFERENCE 6.3

Default Buttons

Gadget Factory automatically provides **Execute** and **Cancel** buttons for gadgets that use its standard dialog configuration.

These buttons integrate with Vectric's native dialog completion mechanism, allowing the programmer to concentrate on the gadget's functionality rather than manually implementing its closing behavior.

Two utility functions provide control over these buttons:

- `GF.ExecutionEnabled()` - Enables, disables, or reports the state of the standard Execute button.
- `GF.NoDefaultButtons()` - Suppresses both default buttons when the gadget provides its own controls.

Neither function requires loading a component library.

Default behavior

Unless instructed otherwise, Gadget Factory appends two buttons to the generated HTML document:

```HTML
<input type="button" id="ButtonNotOK" value="Execute">
<input type="button" id="ButtonCancel" value="Cancel">
```

Notice that the Execute button initially uses the ID `ButtonNotOK` and does not have the `LuaButton` class.

Using `ButtonNotOK` instead of the special `ButtonOK` identifier leaves it without native Execute completion behavior until the gadget enables execution.

The Cancel button, however, is active immediately.

The buttons are part of Gadget Factory's standard HTML template and are added after the content supplied by the gadget.

Dialog completion

Vectric recognizes special button identifiers that determine how `HTML_Dialog:ShowDialog()` completes.

Button ID	Function
`ButtonOK`	Completes the dialog successfully.
`ButtonCancel`	Cancels the dialog.
`ButtonNotOK`	An ordinary button without Vectric's Execute completion behavior.

Gadget Factory uses the result of `ShowDialog()` to determine which processing callback to invoke.

Conceptually:

```lua
local ok = Dlg:ShowDialog()

if ok then
    if Def.ProcessResults then Def.ProcessResults() end
elseif Def.DiscardResults then
    Def.DiscardResults()
end
```

When Execute completes the dialog successfully, Gadget Factory invokes the gadget's `ProcessResults()` callback, if provided.

When the dialog is canceled, it invokes `DiscardResults()`, if provided.

Both callbacks are registered through GF.GadgetDef (p. 10).

The programmer does not need to register ordinary event handlers for the default Execute and Cancel buttons. Their behavior is handled through Vectric's dialog completion mechanism.

Native completion and LuaButton callbacks

The special completion identifiers and ordinary LuaButton callbacks serve different purposes.

In a native Aspire test using Hello-Millie, we called `GF.NoDefaultButtons()` and inserted raw HTML buttons with the identifiers `ButtonOK` and `ButtonCancel`, **without the** `LuaButton` **class**.

Both buttons worked correctly. In particular, `ButtonOK` completed the dialog and Gadget Factory invoked `ProcessResults()`.

In the tested Aspire environment, the special completion identifiers were sufficient. The `LuaButton` class is not universally required for native `ButtonOK` or `ButtonCancel` completion. Gadget Factory's current standard completion-button markup also omits the class.

This result comes from raw HTML. It does not by itself verify a particular ButtonGroup or ToolStrip layout, or establish the same behavior in every Vectric environment.

Ordinary controls that dispatch Lua callbacks still use the LuaButton event mechanism (p. 320). Keep the required callback markup and handler registration for those controls. The completion-button test does not establish that `LuaButton` is unnecessary for ordinary callbacks.

ExecutionEnabled

`GF.ExecutionEnabled()` controls whether the standard Execute button can complete the dialog.

It also provides a convenient way to query the current execution-enabled state.

Syntax

```lua
GF.ExecutionEnabled(state)
```

Parameters

Parameter	Description
`state`	Optional Boolean indicating whether Execute should be enabled.

Return value: Boolean representing the current execution-enabled state.

The parameter has three common uses:

Call	Behavior
`GF.ExecutionEnabled(true)`	Enable Execute.
`GF.ExecutionEnabled(false)`	Disable Execute.
`GF.ExecutionEnabled()`	Return the current state without changing it.

The initial state is `false`.

Basic usage

Suppose a gadget requires the user to select a valid item before execution.

After the dialog has opened, the application can enable Execute when a valid selection is made:

```lua
GF.ExecutionEnabled(true)
```

If the selection becomes invalid:

```lua
GF.ExecutionEnabled(false)
```

The current state can also be queried:

```lua
local enabled = GF.ExecutionEnabled()
```

This returns the state maintained internally by Gadget Factory.

Example: Enabling execution from a checkbox

A simple example illustrates how a control can determine whether execution is permitted.

During InitializeScript() (p. 34), create a checkbox and register its native binding:

```lua
GF.AddElements {
    [[<input type="checkbox" class="LuaButton" id="chkReady">]],
    [[<label for="chkReady">Ready to execute</label>]]
}

GF.AddBinding(function(dlg)
    dlg:AddCheckBox("chkReady", false)
end)
```

Then register the event handler:

```lua
GF.AddHandler("chkReady", function()
    local ready = GF.Dialog:GetCheckBox("chkReady")
    GF.ExecutionEnabled(ready)
end)
```

When the user checks the box, Gadget Factory enables Execute.

When the box is unchecked, Execute becomes inactive again.

This demonstrates how the standard dialog controls can respond to application state without requiring the programmer to recreate or manage the buttons directly.

When can ExecutionEnabled be called?

Although the execution-enabled state can be queried at any point after Gadget Factory is loaded, **changing that state requires the native dialog and its HTML document to be available**.

In particular, do not rely on calling:

```lua
GF.ExecutionEnabled(true)
```

from `InitializeScript()` or `InitializeDialog()`.

`InitializeScript()` runs before the dialog object exists. During `InitializeDialog()`, the native dialog and its field bindings already exist, but DOM modification cannot be relied upon.

In native Aspire verification, calling `GF.ExecutionEnabled(true)` from `InitializeDialog()` failed to enable the Execute button. `ExecutionEnabled()` uses `SetOuterHtml()` to replace the button's markup. The result establishes unreliable modification at this stage, without conclusively proving that the DOM is absent.

Call `ExecutionEnabled()` from a callback after the HTML document has loaded, either a post-load initialization callback or a later user-interaction callback.

See Post-load initialization (p. 367) for the timing mechanism.

An initial call to `ExecutionEnabled(false)` is unnecessary because the default state is already disabled.

How ExecutionEnabled works

Gadget Factory does not use the ordinary HTML `disabled` attribute to control its Execute button.

Instead, it changes the button's entire HTML representation using Vectric's `SetOuterHtml()` method.

When execution is enabled, Gadget Factory replaces `ButtonNotOK` with:

```HTML
<input type="button" id="ButtonOK" value="Execute">
```

When execution is disabled, it replaces `ButtonOK` with:

```HTML
<input type="button" id="ButtonNotOK" value="Execute">
```

The distinction is functional.

The active version uses Vectric's special `ButtonOK` identifier. The inactive version uses `ButtonNotOK`, which does not request native dialog completion.

Both versions omit `LuaButton` in the current implementation. The Hello-Millie test (p. 289) confirms that raw completion buttons can work without the class in the tested Aspire environment.

This means the disabled Execute button remains visible, but activating it does not complete the dialog.

The actual implementation

The complete mechanism is small enough to include here:

```LUA
GF.ExecutionEnabled = (function()
    local Enabled = false

    return function(state)
        if state == nil then return Enabled end

        state = not not state
        if state == Enabled then return Enabled end

        if state then
            Dlg:SetOuterHtml("ButtonNotOK",
                [[<input type="button" id="ButtonOK" value="Execute">]])
        else
            Dlg:SetOuterHtml("ButtonOK",
                [[<input type="button" id="ButtonNotOK" value="Execute">]])
        end

        Enabled = state
        return Enabled
    end
end)()
```

There are several details worth noting.

The function maintains its own state.

The local variable `Enabled` is retained through a Lua closure, using the same general technique discussed in NumberFormatter (p. 281).

Repeated state requests are ignored.

If Execute is already enabled, another call to `ExecutionEnabled(true)` returns immediately without performing an unnecessary HTML replacement.

The function doubles as a getter.

When called without an argument, it returns `Enabled` without modifying the dialog.

Arguments are converted to Boolean values.

The expression:

```lua
state = not not state
```

converts the supplied value to a Lua Boolean.

Remember that Lua considers everything except `false` and `nil` to be true - including zero and empty strings.

For clarity, supplying explicit Boolean values is preferable.

An important distinction

`ExecutionEnabled()` reports the state maintained by Gadget Factory, not a fresh inspection of the HTML document.

If the programmer independently modifies the Execute button's HTML, the function's internal state may no longer accurately describe the displayed control.

For that reason, allow Gadget Factory to manage the standard Execute button rather than replacing it manually.

NoDefaultButtons

`GF.NoDefaultButtons()` prevents Gadget Factory from adding its standard Execute and Cancel buttons to the generated dialog.

It is useful when a gadget provides its own controls, such as a custom button group, toolbar, or specialized action panel.

Syntax

```lua
GF.NoDefaultButtons()
```

Parameters: None.

Return value: None.

Basic usage

Call the function during InitializeScript() (p. 34):

```lua
local function InitializeScript(job)
    GF.NoDefaultButtons()

    -- Construct the gadget's custom interface
end
```

Gadget Factory will omit its normal Execute and Cancel buttons when assembling the HTML document.

This does not automatically provide replacement controls.

The programmer is responsible for constructing any buttons required by the custom interface.

Example: Sectio Divina

Sectio Divina replaces the ordinary Execute and Cancel arrangement with a vertical stack of five actions.

The following code comes from its initialization routine:

```lua
GF.ButtonGroup.New(5, 1, "sdButtons")
    :WithContent({
        {"Revert",  btnRevert},
        {"Undo",    btnUndo},
        {"Apply",   btnApply},
        {"Commit",  "ButtonOK"},
        {"Discard", "ButtonCancel"}
    }, 370, 5)
```

The button group provides three application-specific operations:

- Revert the dimensions to their original values.
- Undo the previous transformation.
- Apply the current dimensions.

It also provides two dialog-completion operations:

- **Commit** uses the special `ButtonOK` action.
- **Discard** uses the special `ButtonCancel` action.

Later in `InitializeScript()`, Sectio Divina calls:

```lua
GF.NoDefaultButtons()
```

This prevents Gadget Factory from appending another pair of Execute and Cancel buttons.

The custom button group takes their place.

Preserving native completion behavior

Using the special Vectric button identifiers allows a custom interface to retain normal dialog-completion semantics.

For example, a custom Commit button that uses `ButtonOK` can complete the dialog successfully, causing Gadget Factory to invoke `ProcessResults()`.

A Discard button using `ButtonCancel` can cancel the dialog, invoking `DiscardResults()` if one is registered.

This is preferable to reproducing Gadget Factory's entire dialog-completion sequence inside ordinary LuaButton handlers.

The Hello-Millie test (p. 289) confirmed completion with raw HTML buttons that omitted `LuaButton`. ButtonGroup and ToolStrip also omit that class for string actions using these IDs. The raw-HTML result supports this use of the identifiers, but does not replace a native exercise of the particular component-generated layout.

What NoDefaultButtons does not do

`NoDefaultButtons()` does not:

- Disable custom buttons.
- Remove controls already added to the gadget's own HTML.
- Alter the `ProcessResults()` or `DiscardResults()` callbacks.
- Register replacement buttons.
- Change the native dialog's completion rules.

It simply prevents Gadget Factory's standard button markup from being included in the generated document.

The current public API does not provide a corresponding function for restoring those default buttons after they have been suppressed.

Consequently, this decision should ordinarily be made during interface initialization.

Combining the two functions

`ExecutionEnabled()` and `NoDefaultButtons()` control different aspects of the standard dialog interface.

Function	Purpose
`ExecutionEnabled()`	Control the functional state of Gadget Factory's standard Execute button.
`NoDefaultButtons()`	Prevent Gadget Factory from generating its standard Execute and Cancel buttons.

These functions are not generally intended to be used together.

If the default buttons have been suppressed, `ExecutionEnabled()` has no standard Execute button to manipulate.

For custom button groups, manage their enabled states through the facilities provided by the component or through the appropriate HTML mechanisms.

The implementation of `ExecutionEnabled()` specifically expects the standard `ButtonOK` and `ButtonNotOK` representations, so it should not be assumed to manage arbitrary custom controls.

Styling the default buttons

Gadget Factory's default buttons are ordinary HTML elements and can be styled using CSS.

For example:

```lua
GF.AddStyle([[
    #ButtonCancel {
        width: 100px;
        height: 35px;
    }

    #ButtonOK, #ButtonNotOK {
        width: 100px;
        height: 35px;
    }
]])
```

Notice that both `ButtonOK` and `ButtonNotOK` are included.

Because `ExecutionEnabled()` switches between these two IDs, styling only one may cause the button's appearance to change when its state changes.

Styling these completion buttons by ID or through a containing element does not require adding a `LuaButton` class.

See AddStyle (p. 108) for additional information about gadget-specific CSS.

Important considerations

- **Initial Execute state:** The standard Execute button is functionally inactive when the dialog is first created.
- **Cancel availability:** The default Cancel button is active immediately.
- **Native completion:** Raw `ButtonOK` and `ButtonCancel` buttons worked without `LuaButton` in the tested Aspire environment. This result does not establish that ordinary callback controls can omit the class.
- **ExecutionEnabled timing:** Change the execution state only after the dialog's document is available for dynamic HTML manipulation.
- **Internal state:** `ExecutionEnabled()` maintains a private Boolean state and does not inspect the current HTML when queried.
- **Repeated calls:** Requests that do not change the state do not trigger HTML replacement.
- **Default-button dependency:** `ExecutionEnabled()` is designed for Gadget Factory's standard Execute button, not arbitrary custom buttons.

- **NoDefaultButtons timing:** Call it during interface construction, before the document is generated.
- **Both buttons suppressed:** `NoDefaultButtons()` removes the standard Execute and Cancel pair, not just Execute.
- **Custom completion controls:** Use the appropriate native button identifiers when custom buttons must complete or cancel the dialog.
- **LuaButton callback styling:** For controls using the LuaButton callback mechanism, do not add other CSS classes to the `LuaButton` class attribute. Use IDs or surrounding containers for styling.
- **Application processing:** Gadget Factory invokes `ProcessResults()` after successful dialog completion and `DiscardResults()` after cancellation, when those callbacks are provided.

Related functions

- GadgetDef (p. 10) - Define the gadget's lifecycle and processing callbacks.
- InitializeScript (p. 34) - Construct the interface and configure default buttons.
- InitializeDialog (p. 49) - Access the native dialog during initialization.
- Dynamic HTML (p. 364) - Understand runtime HTML replacement and document availability.
- Event Handling (p. 320) - Register ordinary LuaButton callback handlers.
- AddStyle (p. 108) - Apply custom styling to dialog controls.
- ButtonGroup (p. 156) - Construct collections of custom action buttons.

Summary

Gadget Factory provides a standard Execute and Cancel button pair that integrates directly with Vectric's native dialog completion mechanism.

`GF.ExecutionEnabled()` controls whether the standard Execute button can complete the dialog, while `GF.NoDefaultButtons()` removes both standard buttons when the gadget supplies its own controls.

These functions allow programmers to use a conventional dialog workflow when appropriate, without preventing the construction of more specialized interfaces.

Use the default buttons when they meet the gadget's needs. Replace them when the interface calls for something different.

REFERENCE 6.4

StatusBar

`GF.StatusBar` provides a persistent information and status display along the bottom of a Gadget Factory dialog.

It contains two independently managed text areas, positioned at the left and right ends of the bar, together with a background indicator that can display three visual states: **Normal**, **Caution**, and **Warning**.

The StatusBar is intended to provide information and feedback without interrupting the user's interaction with the dialog.

Although loaded through Gadget Factory's component-loading mechanism, StatusBar is a **singleton utility**. Unlike ordinary components, it does not support multiple independent instances or use a `New()` constructor.

Loading and initialization

StatusBar is supplied by its own library module and must be loaded before use.

```lua
GF.LoadComponents {"StatusBar"}
```

The module exposes the singleton as `GF.StatusBar`.

Initialize it using:

```lua
GF.StatusBar.Init {
    LeftFormat  = ...,
    GetLeft     = ...,
    RightFormat = ...,
    GetRight    = ...
}
```

All four configuration properties are optional.

Return value: The StatusBar singleton itself.

The returned value may be assigned to a local variable for convenience, although this is not necessary when accessing it through `GF.StatusBar`.

Configuration properties

Property	Description
`LeftFormat`	Optional Lua format string for the left text area.
`GetLeft`	Static value or function supplying the left text.
`RightFormat`	Optional Lua format string for the right text area.
`GetRight`	Static value or function supplying the right text.

The left and right areas are independent. Either may be used without configuring the other.

Basic usage

The simplest configuration displays fixed text.

```lua
GF.LoadComponents {"StatusBar"}

GF.StatusBar.Init {
    GetLeft  = "Ready",
    GetRight = "Gadget Factory"
}
```

This creates a StatusBar with `Ready` aligned to the left and `Gadget Factory` aligned to the right.

The default appearance is determined by the companion `StatusBar.css` stylesheet, which is automatically included when the module is loaded.

No additional HTML, CSS, or field-binding code is required.

Using only one side

Either side may be omitted:

```lua
GF.StatusBar.Init {
    GetLeft = "Ready"
}
```

The bar is still created, but no Vectric label binding is registered for the unconfigured right side.

A side should therefore be configured during initialization if it will need to be updated later.

Dynamic values

For information that changes during execution, supply a function instead of a static value.

```lua
local Message = "Ready"

GF.StatusBar.Init {
    GetLeft = function()
        return Message
    end
}
```

The function is known as a *getter*: its purpose is to retrieve the current value whenever StatusBar needs to refresh that area.

Unlike the function factory discussed in NumberFormatter (p. 281), a StatusBar getter is not expected to create another function. It simply returns the information to display.

The value can be changed elsewhere in the gadget:

```lua
Message = "Processing..."
GF.StatusBar.Update("Left")
```

The first statement changes the application variable.

The second instructs StatusBar to evaluate its getter again and update the visible text.

Changing the variable alone does not automatically update the display.

Static values versus getters

A static value is suitable for information that will remain unchanged.

```lua
GetLeft = "Ready"
```

A getter is preferable when the information can change.

```lua
GetLeft = function()
    return Message
end
```

The distinction is important because `StatusBar.Update()` evaluates the stored configuration. It does not inspect unrelated Lua variables automatically.

For dynamic content, the getter must reference the current application state.

Formatting displayed values

The optional `LeftFormat` and `RightFormat` properties provide access to Lua's `string.format()` functionality.

For example:

```lua
GF.StatusBar.Init {
    LeftFormat = "X: %s, Y: %s",

    GetLeft = function()
        return X, Y
    end
}
```

The getter returns two values, which are passed to the format string.

If `X` is `12.5` and `Y` is `8`, the resulting text is:

```text
X: 12.5, Y: 8
```

Multiple return values

An important Lua convention makes this possible.

The StatusBar implementation performs the equivalent of:

```lua
return fmt:format(get())
```

Because `get()` appears as the final argument expression, Lua passes all of its returned values to `string.format()`.

Consequently, a getter can supply several values to one format string.

```lua
GetLeft = function()
    return X, Y, Z
end
```

These can be used with:

```lua
LeftFormat = "X: %s  Y: %s  Z: %s"
```

This provides a concise way to display several related values without manually concatenating strings.

Combining StatusBar with NumberFormatter

For consistent numeric formatting, StatusBar getters can use NumberFormatter (p. 281).

```lua
local D3 = GF.NumberFormatter(3)

GF.StatusBar.Init {
    LeftFormat = "X: %s  Y: %s",

    GetLeft = function()
        return D3(X), D3(Y)
    end
}
```

This displays each number using a maximum of three decimal places, without unnecessary trailing zeros.

For example:

```text
X: 12.346  Y: 8.5
```

This arrangement separates three responsibilities:

- The application maintains the actual numeric values.
- `D3()` converts them into formatted strings.
- StatusBar combines those strings into the final display.

Formatting without a format string

A getter does not require a corresponding format string.

```lua
GF.StatusBar.Init {
    GetRight = function()
        return Message
    end
}
```

In this case, the getter's first returned value is used directly.

If a getter returns `nil` without a format string, StatusBar substitutes an empty string.

If a format string is provided, the programmer is responsible for ensuring that the getter returns values appropriate for that format.

Standard Lua formatting rules apply, including using `%%` to represent a literal percent sign.

Updating the StatusBar

Once the dialog is running, the displayed values can be refreshed through `StatusBar.Update()`.

Syntax

```lua
GF.StatusBar.Update(side)
```

Parameters

Parameter	Description
`side`	Optional string: `"Left"` or `"Right"`. Omit to update both sides.

Return value: None.

Updating one side

```lua
GF.StatusBar.Update("Left")
```

This evaluates `GetLeft` and updates the corresponding native label field.

Likewise:

```lua
GF.StatusBar.Update("Right")
```

refreshes only the right side.

The names are case-sensitive and should be supplied exactly as shown.

Updating both sides

```lua
GF.StatusBar.Update()
```

When no argument is supplied, StatusBar updates both configured areas.

This is convenient when a single operation changes information displayed on both sides.

Example: Processing progress

```lua
local Message = "Ready"
local Completed = 0

GF.StatusBar.Init {
    GetLeft = function()
        return Message
    end,

    RightFormat = "%d operations completed",

    GetRight = function()
        return Completed
    end
}
```

During execution:

```lua
Completed = Completed + 1
Message = "Processing..."

GF.StatusBar.Update()
```

Both values are refreshed.

The example illustrates updating ordinary application state first, then instructing StatusBar to display it.

When to update

`Update()` requires the native dialog to exist and its bound label fields to be available.

It is normally called from a runtime callback after the dialog has opened.

Do not use `StatusBar.Update()` during `InitializeScript()`. The required Vectric dialog object does not exist at that stage.

Initial values are established automatically during StatusBar initialization and native binding.

Visual status states

In addition to displaying text, StatusBar provides three methods for changing its background indication.

Syntax

```lua
GF.StatusBar.Normal(update)
GF.StatusBar.Caution(update)
GF.StatusBar.Warning(update)
```

Parameters

Parameter	Description
`update`	Optional Boolean. When true, refresh both text areas before changing the visual state.

Return value: None.

The three states use the following default colors:

State	Background	Intended meaning
Normal	<text color="#24734d">Mint cream</text>	Ordinary operation
Caution	<text color="#967400">Lemon chiffon</text>	Something requires attention
Warning	<text color="#a94444">Misty rose</text>	A warning or problem

These meanings are conventions rather than enforced behavior.

StatusBar does not examine the gadget's state or decide which warning level is appropriate. The programmer selects the desired indication.

Changing the state

```lua
GF.StatusBar.Normal()
```

Restores the normal appearance.

```lua
GF.StatusBar.Caution()
```

Changes the background to its caution appearance.

```lua
GF.StatusBar.Warning()
```

Changes the background to its warning appearance.

These calls do not automatically change the displayed text.

Changing state and updating text together

Passing `true` requests a text refresh before changing the visual state.

```lua
Message = "Template not selected"

GF.StatusBar.Caution(true)
```

This is equivalent in effect to:

```lua
Message = "Template not selected"

GF.StatusBar.Update()
GF.StatusBar.Caution()
```

The optional argument provides a convenient way to combine the two operations.

All three state methods accept this argument.

Returning to normal

After correcting a problem:

```lua
Message = "Ready"

GF.StatusBar.Normal(true)
```

The text is refreshed, and the normal background is restored.

This creates a straightforward convention for communicating application state without interrupting the user.

Example: Validation feedback

Consider a gadget that requires positive X and Y dimensions before continuing.

```lua
local Message = "Ready"

local CheckDimensions = function()
    if X <= 0 or Y <= 0 then
        Message = "Dimensions must be greater than zero"
        GF.StatusBar.Warning(true)
        return false
    end

    Message = "Dimensions are valid"
    GF.StatusBar.Normal(true)

    return true
end
```

The function performs validation, updates the application message, and selects the appropriate visual state.

The StatusBar handles refreshing the text and changing its background.

This arrangement is preferable to displaying repeated modal message boxes for routine validation feedback.

During Gadget Factory's development, experiments showed that displaying `MessageBox()` from callbacks could interfere with queued interface updates.

StatusBar offers a less disruptive alternative for information that does not require an immediate modal response.

See Dynamic HTML (p. 364) for additional discussion of Vectric's callback and rendering behavior.

A complete initialization example

The following example demonstrates loading StatusBar, configuring its content, and updating it from a gadget callback.

LUA

```lua
local D3 = GF.NumberFormatter(3)

local X = 12.5
local Y = 8.0
local Message = "Ready"

local function InitializeScript(job)

    GF.LoadComponents {"StatusBar"}

    GF.StatusBar.Init {
        LeftFormat = "X: %s  Y: %s",

        GetLeft = function()
            return D3(X), D3(Y)
        end,

        GetRight = function()
            return Message
        end
    }

    GF.AddElement([[
        <input type="button" class="LuaButton"
               id="btnCheck" value="Check Dimensions">
    ]])

    GF.AddHandler("btnCheck", function()
        if X > 0 and Y > 0 then
            Message = "Dimensions are valid"
            GF.StatusBar.Normal(true)
        else
            Message = "Invalid dimensions"
            GF.StatusBar.Warning(true)
        end
    end)
end
```

The StatusBar is initialized before the dialog is constructed.

Its getters retain access to the gadget's application variables.

When the user activates the button, the registered callback examines those values and updates the StatusBar accordingly.

No additional Vectric field registrations, HTML manipulation, or event handlers are required for the StatusBar itself.

The example assumes the remaining gadget lifecycle is defined through GF.GadgetDef (p. 10).

Preparing HTML independently

StatusBar exposes two additional methods that separate its configuration and HTML generation from normal component registration.

PrepareContent

```lua
GF.StatusBar.PrepareContent(Definition)
```

Accepts the same configuration table as `Init()`.

It stores the formatting options and getter functions, evaluates the initial left and right display values, and retains those values for subsequent HTML generation.

Return value: The StatusBar singleton.

Unlike `Init()`, `PrepareContent()` does not register the StatusBar with Gadget Factory or establish native field bindings.

GetHtml

```lua
local html = GF.StatusBar.GetHtml()
```

Returns the complete StatusBar HTML using the values previously established by `PrepareContent()`.

It does not evaluate the getter functions again, append HTML to the dialog, or register native Vectric fields.

`PrepareContent()` must be called before `GetHtml()`.

Example

```lua
GF.StatusBar.PrepareContent {
    GetLeft  = "Ready",
    GetRight = "Gadget Factory"
}

local html = GF.StatusBar.GetHtml()
```

The returned string contains the StatusBar's complete HTML structure, including its left and right text areas and visual-state element.

This separation allows the markup to be generated independently of the normal dialog-construction process.

Getter evaluation

During ordinary `StatusBar.Init()` processing, getter functions are evaluated when `PrepareContent()` establishes the initial display values.

The subsequent `GetHtml()` call uses those prepared values without invoking the getters again.

Getters are evaluated again during native field binding and later whenever `StatusBar.Update()` refreshes their respective areas.

Consequently, **getter functions should retrieve information without producing unintended side effects**.

For normal gadget development, `StatusBar.Init()` remains the preferred interface. It performs preparation, HTML generation, and registration automatically.

How StatusBar works internally

StatusBar is implemented as a singleton table exported from its library module.

The module registers its companion stylesheet through Gadget Factory's internal component facilities.

Its initialization function constructs the HTML and registers the singleton for native binding.

`Init()` delegates HTML preparation to the two preparation functions before calling `Core.RegisterContent()`

The generated HTML

The essential structure is:

```HTML
<div id="divStatus" class="statusNormal">
    <span id="sbarState" class="statusNormal"></span>
    <span id="slabLeft" class="cssStatusBar">...</span>
    <span id="slabRight" class="cssStatusBar">...</span>
</div>
```

Each element has a distinct purpose:

Element	Purpose
`divStatus`	Main StatusBar container
`sbarState`	Background state indicator
`slabLeft`	Left-aligned text display
`slabRight`	Right-aligned text display

The text spans are registered as Vectric LabelFields when their corresponding getters are configured.

Native field binding

During dialog initialization, Gadget Factory calls the singleton's `Bind()` method.

The relevant implementation is:

```lua
function StatusBar:Bind(dialog)
    dlg = dialog

    for _, Side in ipairs(Sides) do
        if StatusBar["Get" .. Side] then
            dlg:AddLabelField(
                "slab" .. Side,
                GetField(Side)
            )
        end
    end
end
```

This illustrates several previously documented mechanisms.

Deferred binding: The StatusBar does not attempt to bind its label fields until Gadget Factory has created the native dialog.

Conditional registration: Only sides with a configured getter or static value receive native label bindings.

Dynamic member names: Lua constructs property names such as `GetLeft` and `GetRight` by concatenating strings.

This avoids maintaining separate binding logic for the two text areas.

The same technique is used during updates.

How Update works

The internal update function is similarly compact:

```lua
local Update = function(side)
    if StatusBar["Get" .. side] then
        dlg:UpdateLabelField(
            "slab" .. side,
            GetField(side)
        )
    end
end
```

The function retrieves the current value and updates the corresponding native LabelField.

When the public `StatusBar.Update()` function is called without an argument, it invokes this logic for both sides.

This is why the StatusBar does not need special methods for updating its text.

It uses Vectric's existing LabelField mechanism.

How the visual states work

The background-state mechanism is another application of Dynamic HTML (p. 364).

The essential implementation is:

```lua
local SetState = function(state, update)
    if update then StatusBar.Update() end

    dlg:SetOuterHtml(
        "sbarState",
        ('<span id="sbarState" class="status%s"></span>'):format(state)
    )
end
```

Changing the state replaces the `sbarState` element with another span using the appropriate CSS class.

The element retains the same ID.

The available classes are:

```css
#sbarState.statusNormal {
    background-color: mintcream;
}

#sbarState.statusCaution {
    background-color: lemonchiffon;
}

#sbarState.statusWarning {
    background-color: mistyrose;
}
```

Thus, `StatusBar.Warning()` effectively replaces the state indicator with:

```html
<span id="sbarState" class="statusWarning"></span>
```

The stylesheet determines the resulting appearance.

This is the same general principle used by AddStateFlag (p. 340): **Lua establishes the state, while CSS determines its visual presentation.**

The difference is that StatusBar changes an element's CSS class through HTML replacement, whereas StateFlag changes a hidden checkbox's checked state.

Styling and layout

The supplied `StatusBar.css` stylesheet positions the StatusBar along the bottom of the dialog.

Its default layout includes:

- A bar approximately 30 pixels high.
- A subtle top border separating it from the main interface.
- Left-aligned text with a 12-pixel inset.
- Right-aligned text with a corresponding inset.
- Three background colors representing the available visual states.

The text areas use a relatively compact font and are vertically centered within the bar.

Because the StatusBar occupies the bottom of the dialog, other absolutely positioned controls should leave sufficient room for it.

The programmer may override its styling through AddStyle (p. 108) if necessary.

For example:

```lua
GF.AddStyle([[
    #divStatus {
        height: 34px;
    }

    #slabLeft, #slabRight {
        font-size: 13px;
    }
]])
```

The StatusBar's stylesheet is loaded automatically with the module. Gadget-specific CSS may supplement or override its rules using the normal CSS cascade.

Formatting and HTML content

The StatusBar's content is inserted into HTML elements, and format strings may include markup intended for display.

For example:

```lua
LeftFormat = "X: <b>%s</b>  Y: <b>%s</b>"
```

However, content containing HTML should be tested in Aspire, particularly when refreshed through native `UpdateLabelField()` calls.

Neither the formatting logic nor the HTML-generation code automatically escapes arbitrary text.

Values originating from external sources should be escaped appropriately before insertion into HTML.

Important considerations

- **Singleton:** StatusBar is intended to be initialized once per gadget dialog. Repeated initialization can produce duplicate HTML elements and bindings.
- **Library loading:** Load the module through `GF.LoadComponents {"StatusBar"}` before calling `GF.StatusBar.Init()`.
- **Dot notation:** The public methods use dot syntax, such as `GF.StatusBar.Update()`, not colon syntax. They do not expect the singleton as an implicit first argument.
- **Initial values:** Getter functions are evaluated during initialization and binding. They should be safe to call more than once and should not depend on side effects.
- **Getter return values:** A getter without a format string should return one display value. With a format string, it may return multiple values.
- **Static values:** Strings may be supplied instead of functions when content does not change.
- **Absent getters:** A side without a configured getter or value does not receive a native label binding.
- **Explicit refresh:** Changing a variable does not automatically refresh the StatusBar. Call `Update()` or a state method with `true`.
- **State changes:** Normal, Caution, and Warning are visual indications. They do not automatically modify application state or determine whether execution is permitted.
- **Native dialog timing:** Updating text or changing the visual state requires the dialog to exist and its HTML to be available.
- **No automatic persistence:** StatusBar's displayed information and state are not automatically saved between gadget executions.
- **No automatic HTML escaping:** Programmer-supplied content and formatted values must be suitable for their HTML context.
- **No built-in getter validation:** Incorrect format strings or incompatible getter return values may cause Lua formatting errors.
- **No return value from Update:** Updating or changing the visual state does not provide a status result or the current display content.
- **No direct user interaction:** StatusBar is a display facility, not an input control. It does not register user-action callbacks.

Related functions

- Default Buttons (p. 288) - Manage the standard Execute and Cancel controls.
- AddLabelSpan (p. 103) - Create an inline Vectric-bound label.
- AddBinding (p. 329) - Understand deferred native field registration.
- AddStyle (p. 108) - Add gadget-specific CSS.
- AddStateFlag (p. 340) - Represent Boolean interface state through CSS.

- Dynamic HTML (p. 364) - Modify existing HTML elements during dialog execution.
- Event Handling (p. 320) - Register callbacks that can update StatusBar.
- InitializeScript (p. 34) - Initialize utilities and construct the interface.
- GadgetDef (p. 10) - Define the gadget's lifecycle.

Summary

`GF.StatusBar` provides a reusable information display with independently configured left and right text areas and three visual status states.

Its content may be static or supplied through Lua getter functions, with optional formatting through `string.format()`.

At runtime, `Update()` refreshes the displayed information, while `Normal()`, `Caution()`, and `Warning()` modify its visual state.

Because StatusBar uses Gadget Factory's existing HTML-generation and native binding machinery, programmers do not need to manage its underlying LabelFields or HTML elements directly.

The application decides what to report. StatusBar handles how and where to display it.

For routine feedback, validation results, and noncritical warnings, it provides a convenient alternative to interrupting the user with modal message boxes.

CHAPTER 07

07

Advanced topics

Bindings, callbacks, rendering, and safe dynamic updates.

IN THIS CHAPTER

Advanced topics: overview 316

Event Handling 320

AddBinding() 329

AddStateFlag() 340

GetElementsHtml() 351

Dynamic HTML 364

REFERENCE 7.1

Advanced topics: overview

Gadget Factory's standard primitives and components provide most of the functionality needed to construct interactive Vectric gadget interfaces. They generate HTML, register Vectric field bindings, and connect controls to Lua callbacks with relatively little effort from the programmer.

However, not every interface can be constructed entirely from predefined controls.

The **Advanced Topics** section explores the lower-level facilities that allow programmers to extend Gadget Factory's normal behavior, implement custom controls, manipulate HTML dynamically, and interact more directly with Vectric's dialog machinery.

These facilities are part of Gadget Factory's public API. They are not undocumented workarounds or private implementation details.

Understanding the three layers

A useful way to approach advanced interface development is to distinguish three related, but independent, mechanisms.

Layer	Responsibility	Examples
HTML	Defines what appears in the dialog	`AddElement`, `AddDiv`, `SetInnerHtml`
Vectric bindings	Connects HTML elements to native dialog fields	`AddBinding`, `Dlg:AddCheckBox`
Event dispatching	Connects user actions to Lua functions	`AddHandler`, `ReplaceHandler`

The standard Field Primitives (p. 93) combine these mechanisms automatically.

Advanced programming becomes necessary when the programmer wants greater control over one or more of them.

For example, a custom HTML button might need an event handler but no native field binding. A dynamically updated display might require an existing binding but no new event handler.

Recognizing these distinctions makes custom interface development considerably easier.

Creating a custom interaction

The following example demonstrates how ordinary HTML can participate in Gadget Factory's callback system without requiring a dedicated component.

```lua
-- Create a bound status display
GF.AddLabelSpan("labStatus", "Ready")

-- Create an ordinary HTML button
GF.AddElement([[
    <input type="button" class="LuaButton"
           id="btnUpdate" value="Update">
]])

-- Connect the button to a Lua function
GF.AddHandler("btnUpdate", function()
    GF.Dialog:UpdateLabelField("labStatus", "Updated")
end)
```

The button is created directly in HTML, while its click handling is delegated to Gadget Factory.

When the user activates the button, Vectric invokes Gadget Factory's dispatcher, which locates and executes the registered Lua function.

The label is updated using its existing Vectric binding.

This simple example illustrates an important principle: **HTML construction, native field binding, and event handling can be used independently or combined as needed.**

Topics covered

The following chapters explore the advanced API, beginning with the most immediately useful capabilities.

Event handling

AddHandler (p. 320)

Register a Lua callback associated with an HTML control. Learn how Gadget Factory centralizes Vectric's LuaButton dispatching and allows custom controls to participate in the same event system used by framework components.

ReplaceHandler (p. 322)

Replace or remove an existing event handler without reconstructing its HTML control. Useful when a control's behavior must change during gadget execution.

Custom bindings

AddBinding() (p. 329)

Register a function that will receive the native `HTML_Dialog` object during initialization. This allows programmers to bind manually constructed HTML elements to Vectric fields without relying exclusively on Gadget Factory's predefined primitives.

AddStateFlag() (p. 340)

Create a hidden, Vectric-bound checkbox whose state can be manipulated programmatically or used by CSS. This provides a convenient mechanism for representing interface states without displaying an ordinary checkbox.

HTML generation and manipulation

GetElementsHtml() (p. 351)

Generate HTML from a structured element collection without executing the normal field-binding operations.

GetDivHtml (p. 356)

Generate a complete HTML `<div>` container as a string, including nested content.

Dynamic HTML (p. 364)

Use Vectric's `SetInnerHtml()` and `SetOuterHtml()` methods to modify an existing interface while the dialog is running. Understand the relationship between dynamic HTML replacement, element identifiers, and existing Vectric bindings.

Specialized framework behavior

ExecutionEnabled (p. 289)

Programmatically enable or disable Gadget Factory's default Execute button by changing its underlying HTML representation.

NoDefaultButtons (p. 293)

Suppress the standard Execute and Cancel controls when a gadget supplies its own dialog actions.

OnError (p. 327)

Customize error reporting for callbacks dispatched through Gadget Factory, including errors intercepted by its protected callback mechanism.

Lifecycle considerations

Advanced features frequently depend on *when* a particular operation occurs.

During InitializeScript() (p. 34), the programmer constructs HTML, registers bindings, and establishes event handlers.

Gadget Factory subsequently generates the HTML document, creates the native dialog, and processes the registered bindings.

Once the dialog exists, InitializeDialog (p. 49) provides an opportunity to perform additional operations that require the actual dialog object.

While the dialog is running, callbacks can read and update native fields or manipulate the displayed HTML.

A common source of difficulty is attempting to perform an operation before the facility it requires exists. For example, a native field cannot be updated through the dialog until that dialog has been created and the field registered.

Understanding this lifecycle is essential when combining custom HTML with native Vectric controls.

Working within Vectric's limitations

Gadget Factory simplifies many of the peculiarities of Vectric's embedded HTML environment, but it does not eliminate them.

Advanced programmers should be particularly mindful of:

- **Unique identifiers:** HTML elements, native field bindings, and callback registrations must refer to the intended controls.
- **Browser compatibility:** The embedded Trident engine does not necessarily support every feature available in modern browsers.
- **Callback behavior:** Event handlers execute through Vectric's Lua callback mechanism, which imposes some constraints on interaction and error handling.
- **Dynamic updates:** Changing an element's HTML is not the same operation as registering a new native field.
- **Initialization order:** HTML construction, binding registration, dialog creation, and runtime updates are distinct stages.

Where Gadget Factory provides an established mechanism, it is generally preferable to use that mechanism rather than reproduce its internal behavior manually.

However, the framework deliberately leaves these lower-level facilities accessible. Programmers are free to construct controls that its standard components do not anticipate.

A final observation

The advanced API is not intended to replace the standard primitives and components.

Its purpose is to make the framework **extensible without requiring modifications to Gadget Factory itself**.

A custom control can be built using ordinary HTML, connected to Vectric through an explicit binding, and integrated into Gadget Factory's event dispatcher - all from the gadget's own source code.

In that respect, the advanced API is less a collection of special features than a toolkit for extending the framework's capabilities.

The standard primitives handle the familiar cases.

The advanced facilities are there for everything else.

REFERENCE 7.2

Event Handling

Gadget Factory provides a centralized event-dispatching system that connects HTML controls to programmer-defined Lua functions.

Vectric's `HTML_Dialog` supports Lua callbacks through HTML elements using the special `LuaButton` class. Gadget Factory simplifies this mechanism by maintaining a registry of event handlers and routing callback requests through a single global dispatcher.

Two public functions manage this registry:

- `GF.AddHandler()` - Registers a new event handler.
- `GF.ReplaceHandler()` - Replaces or removes an existing handler.

This arrangement allows programmers to create interactive HTML controls without defining separate global callback functions for each control.

AddHandler()

`GF.AddHandler()` associates an HTML control identifier with a Lua function that will execute when Vectric reports an event from that control.

Syntax

```lua
GF.AddHandler(id, func, args)
```

Parameters

Parameter	Description
`id`	The ID of the HTML element that generates the event.
`func`	The Lua function to execute when the event occurs.
`args`	Optional argument passed to the function.

Return value: None.

An attempt to register a handler for an ID that already has one raises a Lua assertion error.

Basic usage

First, create an HTML button and define a callback:

```lua
local ShowStatus = function()
    GF.Dialog:UpdateLabelField("labStatus", "Updated")
end
```

During InitializeScript() (p. 34), construct the controls and register the handler:

```lua
GF.AddLabelSpan("labStatus", "Ready")

GF.AddElement([[
    <input type="button" class="LuaButton"
           id="btnStatus" value="Update">
]])

GF.AddHandler("btnStatus", ShowStatus)
```

When the user clicks **Update**, Gadget Factory executes `ShowStatus()`.

The label's displayed value changes from `Ready` to `Updated`.

Notice the division of responsibilities:

- `GF.AddElement()` creates the HTML button.
- The `LuaButton` class identifies it as a Vectric callback control.
- `GF.AddHandler()` associates the button's ID with a Lua function.
- `GF.AddLabelSpan()` creates and binds the label that the callback updates.

The HTML button does not require a separate Vectric field binding. The `LuaButton` mechanism handles its event notification.

Passing arguments

The optional `args` parameter allows a single callback function to serve multiple controls.

```lua
local ButtonClicked = function(action)
    if action == "Apply" then
        ApplyChanges()
    elseif action == "Reset" then
        ResetValues()
    end
end

GF.AddHandler("btnApply", ButtonClicked, "Apply")
GF.AddHandler("btnReset", ButtonClicked, "Reset")
```

Each registration refers to the same function but supplies a different argument.

When `btnApply` is activated, Gadget Factory effectively executes:

```lua
ButtonClicked("Apply")
```

When `btnReset` is activated:

```lua
ButtonClicked("Reset")
```

Important: The dispatcher passes exactly one argument to the registered function.

It does not automatically supply the originating control ID, and it does not unpack argument tables.

If several values are needed, they may be packaged into a Lua table:

```lua
local ProcessAction = function(options)
    Process(options.mode, options.depth)
end

GF.AddHandler("btnProcess", ProcessAction, {
    mode = "Fine",
    depth = 0.025
})
```

Alternatively, a closure can capture additional values without using the `args` parameter.

When no argument is supplied, the callback receives `nil`, which functions declared without parameters can simply ignore.

ReplaceHandler()

`GF.ReplaceHandler()` changes the function associated with an existing handler ID.

It can also remove a handler or establish one for an ID that has not previously been registered.

Syntax

```lua
GF.ReplaceHandler(id, func, args)
```

Its parameters have the same meanings as those of `GF.AddHandler()`.

Return value: None.

Unlike `AddHandler()`, `ReplaceHandler()` does not reject an existing registration.

Replacing a handler

Suppose a button initially invokes `StartProcess()`:

```lua
GF.AddHandler("btnAction", StartProcess)
```

Later, the application needs the same button to invoke `StopProcess()`:

```lua
GF.ReplaceHandler("btnAction", StopProcess)
```

The HTML control remains unchanged. Only its registered Lua function is replaced.

The next event from `btnAction` executes `StopProcess()`.

Because the dispatcher consults the handler registry each time an event occurs, no HTML reconstruction or native rebinding is required.

Removing a handler

Passing `nil` as the function removes the registration:

```lua
GF.ReplaceHandler("btnAction", nil)
```

The HTML button remains visible and continues to generate Vectric callback events, but Gadget Factory no longer has a function associated with its ID.

The dispatcher safely ignores those events.

Removing a handler does not visually disable its control. If the interface must reflect an unavailable action, that state should be managed separately through HTML, CSS, or the appropriate control mechanism.

A removed handler can subsequently be registered again using either `AddHandler()` or `ReplaceHandler()`.

How event dispatching works

Gadget Factory uses a private Lua table to store its registered callbacks:

```lua
local dispatch = {}
```

Each active entry is indexed by the HTML control's ID and contains a function reference and its optional argument.

For example:

```lua
dispatch["btnCalcX"] = {
    func = btnRecalc,
    args = "X"
}
```

This is the conceptual structure of a registration. Programmers do not access the private table directly.

The actual registration functions

The implementation of both public functions is remarkably compact:

```lua
local ReplaceHandler = function (id, func, args)
    dispatch[id] = func and { func = func, args = args } or nil
end

local AddHandler = function (id, func, args)
    assert(not dispatch[id], "Handler already exists: " .. id)
    ReplaceHandler(id, func, args)
end
```

`ReplaceHandler()` assigns or removes an entry in the dispatch table.

`AddHandler()` simply checks that the ID is not already registered before delegating to `ReplaceHandler()`.

There is no separate callback infrastructure for the two functions. They operate on exactly the same registry.

The global Vectric callback

Vectric calls a global function named `OnLuaButton_XXXX()` when an appropriate HTML control generates a LuaButton event.

Gadget Factory defines that function itself.

The current implementation is:

```lua
function OnLuaButton_XXXX(id)
    local handler = dispatch[id]
    if handler and type(handler.func) == "function" then
        local ok, err = pcall(handler.func, handler.args)
        if not ok then
            -- The gadget's error reporter is also an external callback.
            local reported, reportErr = pcall(GF.OnError, err)
            if not reported then
                DefaultOnError(tostring(err) .. "\n\nOnError failed:\n" .. tostring(reportErr))
            end
        end
    end
return true
end
```

Despite its importance, the function performs only a few operations:

1. Receive the control ID from Vectric.
2. Look up the corresponding handler in the private registry.
3. Execute the registered function using `pcall()`.
4. Forward callback errors to `GF.OnError`.
5. Return `true` to Vectric.

If the ID has no handler, or the registered value is not a function, no application callback is executed.

The dispatcher still returns `true`.

The complete event path

```text
User activates an HTML LuaButton
              |
              ▼
Vectric receives the HTML event
              |
              ▼
OnLuaButton_XXXX(controlId)
              |
              ▼
dispatch[controlId]
              |
              ▼
Registered Lua function
              |
              ▼
Optional argument passed to function
              |
              ▼
Application behavior
```

This single mechanism supports ordinary gadget controls, custom HTML buttons, and callback registrations supplied by Gadget Factory components.

Because the factory owns the global dispatcher, individual gadgets ordinarily do not need to define `OnLuaButton_XXXX()` themselves.

Example: Sectio Divina

Sectio Divina provides a practical demonstration of shared callback functions and explicit arguments.

Its interface contains two manual recalculation buttons:

```html
<input type="button" class="LuaButton"
       id="btnCalcX" value="⚡ X"
       style="height:40px;width:60px;">

<input type="button" class="LuaButton"
       id="btnCalcY" value="⚡ Y"
       style="height:40px;width:60px;">
```

Both use the same callback function:

```lua
local btnRecalc = function(XorY)
    if XorY == "X" then
        X = Orientation == 1 and Y / PHI or Y * PHI
        SetDoubleFields(X, nil)
    else
        Y = Orientation == 1 and X * PHI or X / PHI
```

LUA / CONTINUED

```
        SetDoubleFields(nil, Y)
    end
end
```

The registration statements distinguish the two operations:

LUA

```
GF.AddHandler("btnCalcX", btnRecalc, "X")
GF.AddHandler("btnCalcY", btnRecalc, "Y")
```

When the user clicks ⚡ **X**, the callback receives `"X"` and recalculates that dimension.

Clicking ⚡ **Y** invokes the same function with `"Y"`.

The control IDs are used internally for dispatching, while the supplied argument tells the shared callback which operation to perform.

This avoids duplicating nearly identical recalculation functions.

A bound checkbox

Sectio Divina also uses a manually constructed checkbox:

HTML

```
<input type="checkbox" class="LuaButton" id="togAuto">
```

Unlike an ordinary button, this control also needs a Vectric checkbox binding:

LUA

```
GF.AddBinding(function(dlg)
    dlg:AddCheckBox("togAuto", true)
end)
```

Its handler is registered separately:

LUA

```
GF.AddHandler("togAuto", togAuto)
```

This illustrates an important distinction:

A Vectric field binding and a Gadget Factory event handler serve different purposes.

The binding gives the dialog access to the checkbox's value. The handler determines what Lua code executes when the control generates its LuaButton event.

Some controls require both mechanisms. Others, such as ordinary LuaButton action buttons, need only an event handler.

Custom bindings are covered in AddBinding() (p. 329).

Error handling

Gadget Factory executes registered callbacks using Lua's protected-call mechanism:

```lua
local ok, err = pcall(handler.func, handler.args)
```

If the callback succeeds, execution returns normally to the dispatcher.

If it raises a Lua error, the error is captured rather than being propagated directly through Vectric's callback interface.

The dispatcher forwards the error to:

```lua
GF.OnError(err)
```

By default, Gadget Factory reports errors through `DisplayMessageBox()`.

A gadget may provide a custom error reporter through its GF.GadgetDef.OnError (p. 10) setting.

The factory also protects the error-reporting callback with `pcall()`. If that reporter fails, the default reporter is invoked with both error messages.

This arrangement centralizes callback error handling and reduces the need for every registered function to implement its own protected call.

Callback return values

The return value of a registered handler is not passed back to Vectric.

The factory's global `OnLuaButton_XXXX()` function returns `true` independently of whatever the registered callback returns.

Consequently, a function registered through `GF.AddHandler()` does not need to return `true` merely to satisfy Vectric's callback contract.

This behavior applies specifically to handlers executed through Gadget Factory's dispatcher, not necessarily to other Vectric callback mechanisms.

Event handling and the dialog lifecycle

Handlers are normally registered during InitializeScript() (p. 34), when the gadget constructs its HTML interface.

At that stage, the dispatcher records Lua function references. The native dialog need not exist yet.

When Gadget Factory subsequently creates the dialog and displays it, the registered functions are ready to receive events.

Because registration and HTML construction are separate operations, a handler can be replaced during dialog execution without recreating its HTML control.

However, an event handler that reads or updates native dialog fields must wait until those fields have been registered and the dialog exists.

Gadget Factory exposes the active dialog through `GF.Dialog`.

See InitializeDialog (p. 49) for further discussion of operations that require an instantiated dialog.

Important considerations

- **Unique IDs:** Handler IDs must match their corresponding HTML control IDs. Duplicate registrations through `AddHandler()` raise an error.
- **LuaButton class:** The HTML control must use Vectric's `LuaButton` mechanism to generate the expected callback. Registration alone does not make an arbitrary HTML element interactive.
- **Single callback argument:** The optional `args` value is passed as one argument. Tables are not automatically unpacked.
- **No automatic ID argument:** The registered callback receives `args`, not the originating HTML ID, unless that ID was explicitly supplied as `args`.
- **Function references:** Pass the callback function itself, not the result of calling it.
- **Dynamic replacement:** `ReplaceHandler()` changes the next handler invocation without modifying HTML or native field bindings.
- **Handler removal:** Removing a handler suppresses its Lua behavior but does not remove or disable the HTML control.
- **Centralized errors:** Callback errors are directed through `GF.OnError`.
- **Global ownership:** Gadget Factory owns `OnLuaButton_XXXX()`. Gadget code should not redefine this function when using the factory's dispatcher.
- **Callback side effects:** Vectric's embedded dialog has some unusual event-processing behavior. Avoid unnecessary modal message boxes and complex interface updates inside callbacks.

Related functions

- AddBinding() (p. 329) - Register native Vectric field bindings.
- Field Primitives (p. 93) - Create editable controls with automatic change handling.
- AddElement() (p. 72) - Insert custom HTML controls.
- AddElements() (p. 74) - Build interactive interfaces from structured Lua tables.
- AddStateFlag() (p. 340) - Create a hidden checkbox for interface state.
- InitializeScript (p. 34) - Construct the interface and register handlers.
- InitializeDialog (p. 49) - Work with the instantiated dialog.
- GadgetDef (p. 10) - Configure gadget behavior and error reporting.

REFERENCE 7.3

AddBinding()

`GF.AddBinding()` registers a function that will execute when Gadget Factory creates the native Vectric dialog.

Its primary purpose is to connect manually constructed HTML elements to Vectric's `HTML_Dialog` field-management methods, allowing those elements to participate in the same native binding system used by Gadget Factory's standard primitives and components.

Unlike the Field Primitives (p. 93), `AddBinding()` does not generate HTML or establish a particular field type. It simply arranges for programmer-supplied binding code to execute at the appropriate point in the dialog's initialization.

This provides the flexibility to create controls that Gadget Factory's predefined primitives do not directly support.

Syntax

```lua
GF.AddBinding(func)
```

Parameters

Parameter	Description
`func`	A Lua function that accepts the native `HTML_Dialog` object as its argument.

Return value: None.

The supplied function is registered for later execution. It is not called immediately.

Basic usage

Suppose a gadget needs a checkbox that isn't provided by a dedicated Gadget Factory primitive.

First, create the HTML element:

```lua
GF.AddElement([[
    <input type="checkbox" id="chkPreview">
]])
```

Then register its Vectric binding:

```lua
GF.AddBinding(function(dlg)
    dlg:AddCheckBox("chkPreview", true)
end)
```

The binding function receives the actual `HTML_Dialog` object through its `dlg` parameter.

Its call to `dlg:AddCheckBox()` registers the checkbox with Vectric and establishes its initial checked state.

These two operations serve different purposes:

Operation	Responsibility
`GF.AddElement()`	Creates the HTML element that appears in the dialog.
`GF.AddBinding()`	Registers the function that connects the element to Vectric.

Neither operation substitutes for the other.

Creating an HTML checkbox does not automatically register a Vectric checkbox field. Likewise, registering a field does not generate the corresponding HTML markup.

Both are required when the programmer wants an ordinary HTML control to participate in Vectric's native field-management system.

Understanding deferred binding

One of the most important characteristics of `AddBinding()` is that it postpones execution until the native dialog exists.

Consider:

```lua
GF.AddBinding(function(dlg)
    dlg:AddCheckBox("chkPreview", true)
end)
```

When Lua encounters this statement during `InitializeScript()`, it creates an anonymous function and passes that function to `GF.AddBinding()`.

Gadget Factory stores the function for later use.

The call to `dlg:AddCheckBox()` does not occur yet.

Only after the HTML document has been generated and `HTML_Dialog` has been instantiated does Gadget Factory execute the registered function.

This distinction is essential because the native dialog object does not exist while `InitializeScript()` is constructing the interface.

Why use an anonymous function?

Lua treats functions as values. Just like a string or a table, a function can be passed as an argument to another function.

Our example:

```lua
GF.AddBinding(function(dlg)
    dlg:AddCheckBox("chkPreview", true)
end)
```

is equivalent to:

```lua
local BindPreview = function(dlg)
    dlg:AddCheckBox("chkPreview", true)
end

GF.AddBinding(BindPreview)
```

The anonymous form is convenient because most binding functions are short and have no purpose outside their registration.

The named form may be preferable when the binding logic is sufficiently involved to warrant its own function.

The dialog initialization sequence

To understand exactly when bindings execute, consider Gadget Factory's initialization sequence.

After `InitializeScript()` completes:

1. Gadget Factory generates the complete HTML document.
2. It creates the native `HTML_Dialog` object.
3. It executes the registered bindings, supplying the dialog object to each.
4. It assigns the active dialog to `GF.Dialog`.
5. It invokes `InitializeDialog()`, if provided.
6. It displays the dialog through `ShowDialog()`.

The relevant portion of Gadget Factory's actual implementation is:

```lua
Dlg = HTML_Dialog(false, DOM(), Def.Width, Def.Height, Def.Title)

for _, binding in ipairs(bindings) do
    if type(binding) == "function" then
        binding(Dlg)
    else
        binding:Bind(Dlg)
```

LUA / CONTINUED

```
    end
end

GF.Dialog = Dlg

if Def.InitializeDialog then Def.InitializeDialog(Dlg) end
```

Notice that the binding loop executes **before** `GF.Dialog` is assigned.

Consequently, a binding function should use the `dlg` argument it receives rather than attempting to access the dialog through `GF.Dialog`.

LUA

```
-- Correct
GF.AddBinding(function(dlg)
    dlg:AddCheckBox("chkPreview", true)
end)
```

Do not assume that `GF.Dialog` is available inside a binding function.

Once the binding phase has completed, `GF.Dialog` becomes available to the gadget's normal runtime code.

Example: Sectio Divina

Sectio Divina demonstrates a practical application of `AddBinding()` through its automatic dimension-recalculation checkbox.

Because this interface was constructed manually, it combines raw HTML, a native checkbox binding, and a separate event handler.

The following excerpt has been shortened to emphasize those relationships.

Constructing the control

LUA

```
GF.AddElements {
    {GF.AddDiv, 'class="cssToggle" style="left:10px;bottom:10px;"', {
        [[<input type="checkbox" class="LuaButton" id="togAuto">]],
        [[<label for="togAuto" style="width:60px;">Auto</label>]]
    }}
}
```

The checkbox has two important attributes:

- `id="togAuto"` identifies the control for Vectric field access and Gadget Factory event dispatching.
- `class="LuaButton"` allows it to participate in Vectric's Lua callback mechanism.

The label is associated with the checkbox through its HTML `for` attribute.

Registering the binding

Sectio Divina then registers the native checkbox field:

```lua
GF.AddBinding(function(dlg)
    dlg:AddCheckBox("togAuto", true)
end)
```

This gives the checkbox an initial checked value of `true`.

Once the dialog is running, its current state can be retrieved with:

```lua
Dlg:GetCheckBox("togAuto")
```

Sectio Divina uses this capability when deciding whether changes to dimensions or orientation should trigger automatic recalculation:

```lua
if not Dlg:GetCheckBox("togAuto") then return end
```

Registering the event handler

The checkbox also has a separate callback registration:

```lua
GF.AddHandler("togAuto", togAuto)
```

This connects the checkbox's LuaButton event to the gadget's `togAuto` function.

The complete arrangement therefore consists of three independent pieces:

Mechanism	Purpose
HTML checkbox	Provides the visible control.
`AddBinding()`	Allows Vectric to read, initialize, and update its value.
`AddHandler()`	Executes programmer-defined behavior when the control generates an event.

A control does not necessarily require all three.

For example, a checkbox whose value is read only when the user presses Execute may need an HTML element and a Vectric binding, but no separate event handler.

This illustrates the distinction between **data access and event handling**.

The two mechanisms complement each other without being dependent upon one another.

AddBinding versus AddHandler

Although both functions register Lua functions for later execution, their responsibilities and timing differ considerably.

Characteristic	AddBinding	AddHandler
Purpose	Register native dialog fields	Respond to user-generated events
Executes	During dialog initialization	During dialog interaction
Receives	`HTML_Dialog` object	Optional programmer-supplied argument
Frequency	Once during normal dialog creation	Whenever the registered event occurs
Registry	Private binding collection	Private event-dispatch table
Requires a control ID as its own argument	No	Yes

For example:

```lua
GF.AddBinding(function(dlg)
    dlg:AddCheckBox("chkPreview", true)
end)
```

registers the checkbox with Vectric.

By contrast:

```lua
GF.AddHandler("chkPreview", PreviewChanged)
```

registers the behavior to execute when Vectric reports an event from that control.

The first operation establishes native field access. The second establishes application behavior.

See Event Handling (p. 320) for a detailed explanation of Gadget Factory's callback dispatcher.

How Gadget Factory uses bindings internally

The public `AddBinding()` implementation is remarkably simple:

```lua
local AddBinding = function(func)
    bindings[#bindings + 1] = func
end
```

Gadget Factory maintains a private array named `bindings`.

Each call to `AddBinding()` appends a function to that array.

There is no immediate HTML generation, native field registration, or function execution.

Later, during dialog creation, Gadget Factory traverses the collection and executes the registered binding functions.

Bindings used by field primitives

The standard field helpers use the same underlying collection.

For example, when `GF.AddDoubleField()` constructs a control, Gadget Factory generates its HTML and appends a function that will call Vectric's native `AddDoubleField()` method.

The programmer never sees this intermediate registration because the field primitive handles it automatically.

This is why ordinary fields generally do not require explicit `AddBinding()` calls.

Bindings used by components

The collection also contains component instances registered through Gadget Factory's internal component facilities.

When Gadget Factory encounters a function, it invokes that function with the dialog object.

When it encounters a component object, it calls that component's `Bind()` method:

```lua
if type(binding) == "function" then
    binding(Dlg)
else
    binding:Bind(Dlg)
end
```

This allows ordinary binding functions and reusable component objects to participate in a single initialization sequence.

The distinction is transparent to the gadget programmer.

The public `GF.AddBinding()` interface is intended for functions. The component-object registration machinery is handled internally by Gadget Factory.

AddStateFlag uses AddBinding

The relationship is particularly easy to see in the implementation of AddStateFlag() (p. 340).

```lua
GF.AddStateFlag = function(id, value)
    AddElement(StateFlagHtml(id, value))

    AddBinding(function(dlg)
        dlg:AddCheckBox(id, value)
    end)
end
```

The function creates a hidden HTML checkbox and registers its native Vectric binding using the same public `AddBinding()` mechanism available to gadget programmers.

The checkbox can then represent an internal interface state without appearing as an ordinary visible control.

This is a useful illustration of Gadget Factory's design philosophy: relatively simple facilities can be combined to produce more sophisticated behaviors.

Registering multiple fields

A single binding function may contain several native field-registration calls.

For example:

```lua
GF.AddBinding(function(dlg)
    dlg:AddCheckBox("chkPreview", true)
    dlg:AddCheckBox("chkGuides", false)
end)
```

Both HTML checkboxes must exist in the generated dialog document.

Gadget Factory invokes the binding function once, and the function registers both controls.

This is useful when several closely related fields share initialization logic.

For independent controls, separate registrations may be easier to maintain:

```lua
GF.AddBinding(function(dlg)
    dlg:AddCheckBox("chkPreview", true)
end)

GF.AddBinding(function(dlg)
    dlg:AddCheckBox("chkGuides", false)
end)
```

Both approaches are valid.

Binding functions are executed in the order in which they were registered, together with the other bindings collected by Gadget Factory.

Using AddBinding within AddElements

Because `GF.AddBinding` is an ordinary function, it can also appear in a function-call table processed by AddElements() (p. 74).

For example:

```lua
GF.AddElements {
    [[<input type="checkbox" id="chkPreview">]],

    {GF.AddBinding, function(dlg)
        dlg:AddCheckBox("chkPreview", true)
    end}
}
```

`AddElements()` executes the `GF.AddBinding()` call while processing the table.

The binding function itself remains deferred until dialog initialization.

This allows an HTML element and its corresponding registration to be kept together in the same Lua structure.

However, **this pattern is intended for ordinary `AddElements()` processing, not GetElementsHtml()** (p. 351).

`GetElementsHtml()` only supports element functions with registered read-only renderers. `GF.AddBinding()` has no such renderer, because registering a native field is an operation rather than HTML generation.

For most gadgets, registering custom bindings separately from HTML construction remains the clearest approach.

When to use AddBinding

Most programmers will not need `GF.AddBinding()` when using Gadget Factory's standard fields or component libraries.

Those facilities already register their native bindings.

Use `AddBinding()` when manually constructed HTML requires native Vectric field functionality that the available primitives do not provide.

Typical situations include:

- Registering manually constructed checkboxes.
- Binding custom HTML controls to native Vectric fields.
- Grouping related native field registrations.
- Experimenting with specialized interfaces before developing reusable components.

For example, a manually constructed checkbox might use `AddBinding()` to establish native field access, `AddHandler()` to respond to user interaction, and CSS to control its visual appearance.

This arrangement provides access to Vectric's native functionality without modifying Gadget Factory itself.

Important considerations

- **HTML must exist:** The corresponding HTML element must be present in the generated document before native binding takes place.
- **Unique identifiers:** IDs supplied to native field-registration methods must match the intended HTML elements. Avoid registering multiple native fields under the same ID.
- **Deferred execution:** `AddBinding()` registers a function; it does not execute the function immediately.
- **Dialog parameter:** Use the supplied `dlg` argument inside the binding function. `GF.Dialog` is assigned only after the binding phase.
- **Initialization timing:** Register bindings during InitializeScript() (p. 34). Registering additional bindings after the binding phase has completed does not apply them to the active dialog.
- **Registration order:** Binding functions execute in their registration order, alongside component bindings.
- **Function arguments:** The registered function receives the native dialog object. `AddBinding()` does not supply a control ID or other arguments automatically.
- **No automatic events:** Registering a native field does not register a Gadget Factory event handler. Use AddHandler (p. 320) when callback behavior is required.
- **No automatic HTML generation:** The programmer must supply the corresponding markup separately.
- **No automatic validation:** Gadget Factory does not inspect or validate the function or its intended field registrations when `AddBinding()` is called.
- **Binding errors:** Unlike callbacks executed through Gadget Factory's protected event dispatcher, binding functions execute directly during dialog initialization. An error in a binding function is not automatically routed through `GF.OnError`.
- **Prefer standard primitives:** Use Gadget Factory's existing field helpers when they meet the requirement. Manual bindings are most useful when extending their capabilities.

Related functions

- AddHandler (p. 320) - Associate HTML control events with Lua functions.
- ReplaceHandler (p. 320) - Change an existing event handler.
- Field Primitives (p. 93) - Create automatically bound editable and display fields.
- AddStateFlag() (p. 340) - Create a hidden checkbox using Gadget Factory's binding mechanism.
- AddElement() (p. 72) - Insert manually constructed HTML.
- AddElements() (p. 74) - Construct interfaces using nested Lua tables.
- GetElementsHtml() (p. 351) - Generate HTML without executing native field-registration operations.
- InitializeScript (p. 34) - Construct HTML and register bindings.
- InitializeDialog (p. 49) - Access the dialog after native bindings have been established.

Summary

`GF.AddBinding()` bridges manually constructed HTML and Vectric's native field-management system.

Its implementation simply stores a function for later execution, but its placement in Gadget Factory's initialization sequence is what makes it useful.

By separating HTML generation, native field binding, and event dispatching, Gadget Factory allows programmers to combine these mechanisms as needed rather than requiring every control to be implemented through a dedicated primitive or component.

`AddBinding()` connects a control to Vectric. `AddHandler()` connects an event to Lua.

Together, they provide the foundation for extending Gadget Factory beyond its predefined controls.

REFERENCE 7.4

AddStateFlag()

`GF.AddStateFlag()` creates an invisible, Vectric-bound checkbox that represents a Boolean state within the gadget's HTML interface.

Unlike an ordinary checkbox, a StateFlag is not intended for direct user interaction. Instead, its checked state is controlled programmatically and can be used by CSS to change the appearance or visibility of other HTML elements.

This provides a lightweight mechanism for controlling visual states without repeatedly modifying HTML or invoking JavaScript.

A StateFlag combines two Gadget Factory facilities:

- AddElement (p. 72) - Creates the hidden HTML checkbox.
- AddBinding (p. 329) - Registers the checkbox with Vectric's native dialog interface.

Syntax

```lua
GF.AddStateFlag(id, value)
```

Parameters

Parameter	Description
`id`	Unique identifier for the hidden checkbox and its Vectric binding.
`value`	Initial Boolean state: `true` for checked or `false` for unchecked.

Return value: None.

Basic usage

```lua
GF.AddStateFlag("flagReady", false)
```

This creates the following HTML:

```html
<input type="checkbox" id="flagReady" style="display:none;">
```

The checkbox exists in the document but is not displayed because its CSS includes `display:none`.

Gadget Factory also registers the corresponding native Vectric field:

```lua
dlg:AddCheckBox("flagReady", false)
```

The initial value is established through this binding, rather than through an HTML `checked` attribute.

Reading and updating a StateFlag

Once the dialog has been created, its state can be accessed using Vectric's native checkbox methods:

```lua
-- Read the current state
local ready = GF.Dialog:GetCheckBox("flagReady")

-- Set the state
GF.Dialog:UpdateCheckBox("flagReady", true)
```

There are no special StateFlag getter or setter methods.

A StateFlag is an ordinary Vectric checkbox whose HTML representation has been deliberately hidden.

That distinction explains much of its usefulness.

Using CSS to respond to state changes

CSS supports a `:checked` pseudo-class that matches checked checkboxes.

Because a StateFlag is still an HTML checkbox, CSS selectors can examine its state even though the checkbox itself is invisible.

For example:

```css
#flagReady:checked {
    /* Rules applied when the flag is checked */
}
```

However, styling the invisible checkbox itself is rarely useful.

The interesting possibility is using its state to control another element.

Example: Conditional visibility

Consider an interface containing a hidden StateFlag followed by a status message.

```lua
GF.AddStateFlag("flagReady", false)

GF.AddElement([[
    <span id="readyMessage">Ready to execute</span>
]])
```

Add the following CSS:

```lua
GF.AddStyle([[
    #readyMessage {
        display: none;
    }

    #flagReady:checked + #readyMessage {
        display: inline;
    }
]])
```

Initially, the message is hidden because `flagReady` is unchecked.

When the gadget executes:

```lua
GF.Dialog:UpdateCheckBox("flagReady", true)
```

the checkbox becomes checked, and the CSS selector displays the message.

Setting the flag back to `false` hides the message again.

No HTML replacement or JavaScript function is required.

Understanding the CSS selector

The expression:

```css
#flagReady:checked + #readyMessage
```

contains three important parts.

Expression	Meaning
`#flagReady`	The element with ID `flagReady`.
`:checked`	Only match when that checkbox is checked.
`+ #readyMessage`	Select the immediately following sibling element with ID `readyMessage`.

The + operator is known as the *adjacent sibling combinator*.

It requires the two HTML elements to share the same parent, with the target immediately following the StateFlag among element siblings.

Ordinary whitespace or text nodes between them do not prevent the selector from matching, but another HTML element does.

For example:

```HTML
<input type="checkbox" id="flagReady" style="display:none;">
<span id="readyMessage">Ready to execute</span>
```

This structure supports the selector.

However:

```HTML
<input type="checkbox" id="flagReady" style="display:none;">
<div>Other content</div>
<span id="readyMessage">Ready to execute</span>
```

does not, because the intervening `<div>` breaks the required adjacency.

The general sibling selector

CSS also provides the ~ combinator.

```CSS
#flagReady:checked ~ .statusIndicator {
    display: block;
}
```

This matches subsequent sibling elements with the class `statusIndicator`, even if other sibling elements appear between them.

The target elements must still share the same parent as the StateFlag and appear after it.

This allows one StateFlag to control several related elements without requiring them to be immediately adjacent.

Both combinators are useful when constructing state-driven interfaces.

Example: Sectio Divina

Sectio Divina uses a StateFlag to display a small golden-ratio indicator when the current dimensions satisfy its mathematical test.

The flag is named `chkAureity`.

Calculating the state

The gadget defines the following function:

```lua
local CheckAureity = function()
    local IsGolden = false

    if X > 0 and Y > 0 then
        local ratio = math.max(X, Y) / math.min(X, Y)
        IsGolden = math.abs(ratio - PHI) < GOLDEN_TOLERANCE
    end

    if Dlg then
        Dlg:UpdateCheckBox("chkAureity", IsGolden)
    end

    return IsGolden
end
```

The function checks whether the relationship between the X and Y dimensions is sufficiently close to the golden ratio.

It returns a Boolean result and, when the dialog exists, updates the StateFlag.

The test against `Dlg` is important because the same function is used both during initialization and after the dialog has been created.

Creating the flag

During InitializeScript() (p. 34), Sectio Divina creates the StateFlag using the calculated initial value:

```lua
GF.AddStateFlag("chkAureity", CheckAureity())
```

At this point, the dialog does not yet exist, so `CheckAureity()` calculates and returns the initial state without attempting to update the native checkbox.

That value is subsequently supplied to Vectric when Gadget Factory executes the registered binding.

Defining the display behavior

The gadget then adds these CSS rules:

```lua
GF.AddStyle([[
#vaci { position:absolute; left:10px; top:95px; }
#chkAureity + #vaci { display:none; }
#chkAureity:checked + #vaci { display:block; }
]])
```

The element identified as `vaci` is an image representing the golden-ratio indicator.

Its HTML is added immediately after the StateFlag, although the original image's lengthy Base64 data has been omitted from this excerpt:

```lua
GF.AddElement([[
    <img id="vaci" width="30" height="30"
         src="...">
]])
```

The CSS establishes three rules:

1. Position the image at its designated coordinates.
2. Hide the image by default.
3. Display the image whenever `chkAureity` is checked.

Because the StateFlag and image are adjacent siblings in the generated HTML, the CSS selectors work without additional containers or event handlers.

Updating the indicator

After Sectio Divina applies a dimensional transformation, it calls:

```lua
CheckAureity()
```

This recalculates the dimension ratio and updates the checkbox:

```lua
Dlg:UpdateCheckBox("chkAureity", IsGolden)
```

The browser then evaluates the CSS rules against the updated checked state.

The indicator appears when the dimensions meet the golden-ratio criterion and disappears otherwise.

The gadget never explicitly instructs the image to appear or disappear.

It updates the underlying state, and CSS determines the resulting appearance.

This is the principal advantage of the StateFlag mechanism.

How AddStateFlag works

The implementation is short enough to reproduce in its entirety.

Gadget Factory first defines the HTML generator:

```lua
local StateFlagHtml = function(id, value)
    return '<input type="checkbox" id="' .. id ..
           '" style="display:none;">'
end
```

It then exposes the corresponding public functions:

```lua
GF.GetStateFlagHtml = StateFlagHtml

GF.AddStateFlag = function(id, value)
    AddElement(StateFlagHtml(id, value))

    AddBinding(function(dlg)
        dlg:AddCheckBox(id, value)
    end)
end
```

The sequence is straightforward:

1. Generate an invisible HTML checkbox.
2. Append that HTML to the dialog content collection.
3. Register a deferred binding that creates the native Vectric checkbox field.

There is no private StateFlag object, no special state registry, and no dedicated dispatching mechanism.

Gadget Factory relies entirely on existing HTML, CSS, and Vectric functionality.

Why doesn't the HTML contain a checked attribute?

Notice that `StateFlagHtml()` accepts `value` but does not use it in the generated markup.

The initial Boolean value is instead supplied through:

```lua
dlg:AddCheckBox(id, value)
```

Vectric's native binding establishes the checkbox state before the dialog is displayed.

This keeps the StateFlag's HTML generation separate from its state initialization.

StateFlags and event handlers

A StateFlag does not automatically register an AddHandler (p. 320) callback.

It also does not use Vectric's `LuaButton` HTML class.

This is intentional.

The StateFlag is not a visible user control, so it normally has no reason to generate user-interaction events.

The intended sequence is:

```text
Gadget performs an operation
            |
            v
Lua calculates a Boolean state
            |
```

TEXT / CONTINUED

```
            v
Dlg:UpdateCheckBox(id, value)
            |
            v
HTML checkbox changes state
            |
            v
CSS :checked selector responds
            |
            v
Other HTML elements change appearance
```

This differs from ordinary event handling, where an HTML control causes Lua code to execute.

With a StateFlag, **Lua changes a control's state, and CSS responds to the change.**

The flow of control is effectively reversed.

Using AddStateFlag with AddElements

Like the other Gadget Factory primitives, `AddStateFlag()` can be invoked through the structured tables accepted by AddElements() (p. 74).

For example:

LUA

```lua
GF.AddElements {
    {GF.AddStateFlag, "flagReady", false},
    [[<span id="readyMessage">Ready to execute</span>]]
}
```

The function-call table creates and binds the StateFlag, while the following string creates the HTML message.

Both elements are appended sequentially, preserving the sibling relationship needed for the CSS selector.

This is a convenient way to keep the flag and the elements it controls together in the interface definition.

A useful structural consideration

Lua table nesting does not automatically create HTML containers.

For example:

LUA

```lua
GF.AddElements {
    {
        {GF.AddStateFlag, "flagReady", false},
        [[<span id="readyMessage">Ready</span>]]
    }
}
```

The inner table merely groups the two entries for processing. They are still emitted as adjacent HTML elements.

In contrast, using AddDiv() (p. 80) explicitly creates another HTML container.

When writing CSS that depends on sibling relationships, the resulting HTML structure matters more than the nesting of the Lua tables that produced it.

GetStateFlagHtml

Gadget Factory also exposes a read-only HTML generator:

```lua
GF.GetStateFlagHtml(id, value)
```

It returns the HTML checkbox markup without appending it to the dialog or registering a native Vectric field.

For example:

```lua
local html = GF.GetStateFlagHtml("flagReady", true)
```

Returns:

```html
<input type="checkbox" id="flagReady" style="display:none;">
```

Notice that the returned HTML is the same regardless of the supplied Boolean value.

The initial state is not represented in the generated HTML. It is established by the native binding performed by `AddStateFlag()`.

Consequently, `GetStateFlagHtml()` alone does not produce a fully initialized StateFlag.

It is also registered as a renderer for GetElementsHtml() (p. 351), allowing that function to produce StateFlag markup without performing the associated native binding.

Use `AddStateFlag()` for normal interactive dialog construction.

When to use StateFlags

StateFlags are particularly useful when the application needs to communicate simple Boolean conditions to CSS.

Typical uses include:

- Showing or hiding status indicators.
- Revealing contextual instructions.
- Displaying warning symbols when a condition exists.
- Changing colors or other styling based on application state.
- Controlling the appearance of multiple sibling elements through a shared condition.

They are especially appropriate when the desired result is purely visual.

For example, revealing an icon based on a Boolean condition is often simpler with a StateFlag than with repeated calls to `SetInnerHtml()`.

However, StateFlags are not a general replacement for dynamic HTML manipulation.

When the content itself must change, rather than merely its appearance, methods such as `SetInnerHtml()` may be more appropriate.

Important considerations

- **Boolean values:** Supply `true` or `false` as the initial state. Gadget Factory does not validate or convert the value.
- **Unique identifiers:** The StateFlag ID must be unique within the dialog.
- **Invisible, not absent:** The checkbox remains part of the HTML document despite `display:none`.
- **Native binding:** The StateFlag is registered through Vectric's `AddCheckBox()` method.
- **No automatic callbacks:** StateFlags do not register event handlers or use the LuaButton class.
- **Runtime updates:** Use `Dlg:UpdateCheckBox()` after the dialog has been initialized.
- **Native field access:** Use `Dlg:GetCheckBox()` when its current state is needed in Lua.
- **CSS structure:** Sibling selectors depend on the actual HTML element hierarchy and ordering.
- **CSS specificity:** State-driven rules must have sufficient specificity to override the target element's ordinary styling. Inline styles and `!important` declarations may affect the result.
- **Browser compatibility:** State-driven visual effects must use CSS supported by Vectric's embedded browser engine.
- **No automatic persistence:** The flag represents dialog state. Gadget Factory does not automatically save its value between executions.
- **HTML generation:** `GetStateFlagHtml()` generates markup only; it does not establish the initial checked state or native binding.
- **Initialization:** StateFlags should normally be created during InitializeScript() (p. 34), before Gadget Factory creates and binds the dialog.
- **No automatic escaping:** The supplied ID is inserted directly into an HTML attribute and must be suitable for that purpose.

Related functions

- AddBinding (p. 329) - Register native Vectric field bindings.
- AddElement() (p. 72) - Insert manually constructed HTML.
- AddElements() (p. 74) - Construct HTML and StateFlags using structured Lua tables.
- AddDiv() (p. 80) - Create HTML containers.

- AddStyle() (p. 108) - Define CSS rules for StateFlag-driven visual behavior.
- Event Handling (p. 320) - Register callbacks for actual user interaction.
- GetElementsHtml() (p. 351) - Generate HTML without native binding operations.
- Dynamic HTML (p. 364) - Modify displayed content while a dialog is running.
- InitializeScript (p. 34) - Construct the interface and register its bindings.

Summary

`GF.AddStateFlag()` provides a hidden Boolean state that is accessible to both Lua and CSS.

Internally, it is an invisible HTML checkbox registered through Vectric's native binding system. Lua updates its state through `UpdateCheckBox()`, while CSS can respond using the `:checked` pseudo-class.

This creates a useful separation of responsibilities:

Lua decides whether a condition is true. CSS decides how that condition should appear.

Because the mechanism uses existing HTML and Vectric facilities, no additional callback dispatching or specialized state-management machinery is required.

REFERENCE 7.5

GetElementsHtml()

`GF.GetElementsHtml()` generates an HTML string from a structured Lua element collection without executing the element functions normally used to construct a Gadget Factory dialog.

It accepts the same nested-table conventions as AddElements (p. 74), including raw HTML strings, function-call tables, and nested element collections.

However, rather than executing the supplied element functions and registering their associated Vectric bindings, it invokes corresponding **read-only HTML renderers**.

This makes it possible to reuse Gadget Factory's HTML construction syntax when the programmer needs an HTML string rather than immediate dialog construction.

A closely related function, `GF.GetDivHtml()`, provides the same capability for complete HTML `<div>` containers.

Syntax

```lua
local html = GF.GetElementsHtml(elements)
```

Parameters

Parameter	Description
`elements`	An array-style Lua table containing HTML strings, supported element-function tables, or nested collections.

Return value: A string containing the generated HTML markup.

Unlike `GF.AddElements()`, this function does not append anything to Gadget Factory's internal HTML collection.

Basic usage

Consider a simple collection containing an HTML heading and paragraph.

```lua
local elements = {
    "<h3>Dimensions</h3>",
    "<p>Enter the desired dimensions.</p>"
}

local html = GF.GetElementsHtml(elements)
```

The returned string contains:

```HTML
<h3>Dimensions</h3><p>Enter the desired dimensions.</p>
```

The function concatenates the strings in their original order.

Notice that it does not automatically insert line breaks or other separators.

The generated HTML is available for subsequent use, but it has not been added to the dialog.

Comparing AddElements and GetElementsHtml

The difference becomes clearer when both functions receive the same collection.

```LUA
local elements = {
    "<h3>Dimensions</h3>",
    "<p>Enter the desired dimensions.</p>"
}

GF.AddElements(elements)

local html = GF.GetElementsHtml(elements)
```

The first call appends the markup to the dialog's content collection.

The second call generates and returns equivalent markup without adding anything to that collection.

Function	Primary operation	Return value
`AddElements`	Processes elements and performs their associated operations	None
`GetElementsHtml`	Generates HTML without performing those operations	HTML string

For collections containing only strings, the resulting HTML is equivalent.

For collections containing Gadget Factory primitives, an additional distinction becomes important: native bindings and callbacks.

Using element functions

Like `AddElements()`, the function recognizes tables whose first entry is a function.

```LUA
local html = GF.GetElementsHtml {
    {GF.AddDiv, 'class="summary"', {
        "<b>Dimensions:</b> ",
        "12.5 × 8"
    }}
}
```

This generates:

```HTML
<div class="summary"><b>Dimensions:</b> 12.5 × 8</div>
```

Although the collection refers to `GF.AddDiv`, **that function is not actually executed**.

Instead, `GetElementsHtml()` finds its registered rendering counterpart, `GF.GetDivHtml()`, and invokes that function with the supplied arguments.

This distinction is the basis of Gadget Factory's read-only HTML generation.

Nested element collections

Nested tables are handled recursively, using the same conventions as `AddElements()`.

```LUA
local html = GF.GetElementsHtml {
    {GF.AddDiv, 'class="outer"', {
        "<h3>Options</h3>",

        {GF.AddDiv, 'class="inner"', {
            "<p>First option</p>"
        }},

        {GF.AddDiv, 'class="inner"', {
            "<p>Second option</p>"
        }}
    }}
}
```

Each `GF.AddDiv` reference is translated into a call to its HTML renderer.

The resulting string contains the complete nested HTML structure, including the appropriate opening and closing container tags.

As with `AddElements()`, nesting a Lua table does not itself create an HTML container. Only a function such as `GF.AddDiv` explicitly introduces one.

Read-only HTML renderers

Gadget Factory maintains a private table named `ElementRenderers`.

This table associates supported element-construction functions with corresponding functions that generate HTML without performing their normal side effects.

The current factory supports the following mappings:

Element function	HTML renderer
`GF.AddElement`	Returns the supplied HTML string

Element function	HTML renderer
GF.AddElements	GF.GetElementsHtml
GF.AddDiv	GF.GetDivHtml
GF.AddControlLabel	GF.GetLabelHtml
GF.AddLabelSpan	GF.GetLabelSpanHtml
GF.AddLabelField	GF.GetLabelFieldHtml
GF.AddDoubleField	GF.GetDoubleFieldHtml
GF.AddTextField	GF.GetTextFieldHtml
GF.AddIntegerField	GF.GetIntegerFieldHtml
GF.AddStateFlag	GF.GetStateFlagHtml

For example, consider:

```lua
local html = GF.GetElementsHtml {
    {GF.AddLabelSpan, "labStatus", "Ready"}
}
```

Rather than invoking `GF.AddLabelSpan()`, the function calls its registered renderer:

```lua
GF.GetLabelSpanHtml("labStatus", "Ready")
```

The resulting string is:

```html
<span id="labStatus">Ready</span>
```

But no Vectric label binding is registered.

This is precisely what makes the mechanism useful for generating HTML independently of the normal dialog construction process.

HTML generation is not field binding

This is the most important restriction to understand when using `GetElementsHtml()`.

Many Gadget Factory primitives perform two operations during ordinary construction:

1. Generate the appropriate HTML.

2. Register native Vectric bindings, callback handlers, or other initialization operations.

`GetElementsHtml()` performs only the HTML-generation portion.

For example:

```lua
local html = GF.GetElementsHtml {
    {GF.AddDoubleField, "txtDepth", 0.125}
}
```

The resulting HTML includes a text input and the hidden LuaButton used by Gadget Factory's DoubleField change mechanism.

However, this call does not:

- Register a native `Dlg:AddDoubleField()` binding.
- Register the DoubleField's callback dispatcher.
- Establish the field's initial value through Vectric.

The HTML may resemble a fully constructed Gadget Factory field, but its normal Vectric functionality has not been initialized.

Generating a control's HTML does not make that control a working Vectric-bound field.

An existing binding may still be usable when dynamically replacing markup that preserves its registered control ID, but `GetElementsHtml()` does not create new bindings.

An important exception to the idea of being read-only

The term *read-only* describes the rendering of Gadget Factory elements.

It does not prevent ordinary Lua expressions from executing while an element collection is being constructed.

For example:

```lua
local html = GF.GetElementsHtml {
    "<p>" .. CalculateDimensions() .. "</p>"
}
```

Lua calls `CalculateDimensions()` when evaluating the table contents, before `GetElementsHtml()` begins processing them.

The renderer does not suppress side effects in programmer-supplied expressions.

Example: Sectio Divina

Sectio Divina uses `AddElements()` to construct its dimensional history display.

One of its element collections contains:

```lua
{GF.AddDiv, 'style="margin-top:20px;"', {
    [[<b style="margin-right:6px;">Current Dimensions:</b>]],
    {GF.AddLabelSpan, "labX", D(X)}, " × ",
    {GF.AddLabelSpan, "labY", D(Y)},
}},
```

In Sectio Divina, this structure is processed by `GF.AddElements()`, which creates the HTML and registers the label bindings.

The same structure could instead be processed by `GF.GetElementsHtml()`:

```lua
local html = GF.GetElementsHtml {
    {GF.AddDiv, 'style="margin-top:20px;"', {
        [[<b style="margin-right:6px;">Current Dimensions:</b>]],
        {GF.AddLabelSpan, "labX", D(X)}, " × ",
        {GF.AddLabelSpan, "labY", D(Y)},
    }}
}
```

This alternative generates the same HTML representation but does not register `labX` and `labY` as native Vectric label fields.

The example is adapted to demonstrate the distinction; Sectio Divina itself uses `AddElements()` for these controls.

This is an important design choice:

Use `AddElements()` when constructing bound interface controls. Use `GetElementsHtml()` when the generated markup itself is the desired result.

GetDivHtml

`GF.GetDivHtml()` is the read-only counterpart to AddDiv (p. 80).

It creates a complete HTML `<div>` container and returns its markup as a string.

Syntax

```lua
local html = GF.GetDivHtml(attributes, elements)
local html = GF.GetDivHtml(elements)
```

Parameters

Parameter	Description
`attributes`	Optional HTML attribute string for the container.
`elements`	Optional nested element collection defining its contents.

Return value: A string containing the complete `<div>` element, including its opening and closing tags.

Basic usage

```lua
local html = GF.GetDivHtml('class="panel"', {
    "<h3>Options</h3>",
    "<p>Select the desired settings.</p>"
})
```

The resulting HTML is:

```html
<div class="panel"><h3>Options</h3><p>Select the desired settings.</p></div>
```

The actual returned string also includes a newline following the closing `</div>`.

Omitting attributes

As with `GF.AddDiv()`, the attributes argument may be omitted.

```lua
local html = GF.GetDivHtml {
    "<p>Ready</p>"
}
```

This generates an ordinary `<div>` without attributes.

Empty containers

The function can also generate an empty container:

```lua
local html = GF.GetDivHtml('id="placeholder"')
```

Result:

```html
<div id="placeholder"></div>
```

Such a container may be useful when its contents will be supplied dynamically.

How GetDivHtml works

The implementation is straightforward:

```lua
GF.GetDivHtml = function(attributes, elements)
    if type(attributes) == "table" and elements == nil then
        elements   = attributes
        attributes = nil
    end
    return DivHtml(attributes) ..
        (elements and GF.GetElementsHtml(elements) or "") .. "</div>\n"
end
```

The function:

1. Determines whether attributes were supplied.
2. Generates the opening `<div>` tag.
3. Uses `GF.GetElementsHtml()` to render any supplied contents.
4. Appends the closing tag and a newline.
5. Returns the assembled string.

Unlike `GF.AddDiv()`, it does not append the generated markup to Gadget Factory's document content collection.

Dynamic HTML example

One useful application of the read-only rendering functions is generating content for an existing HTML container while the dialog is running.

Suppose a gadget needs to display an updated dimensional summary.

During InitializeScript (p. 34), create an empty container:

```lua
GF.AddDiv('id="details"')
```

Then define a function that generates the desired HTML.

```lua
local D3 = GF.NumberFormatter(3)

local DimensionSummary = function(x, y)
    return GF.GetElementsHtml {
        {GF.AddDiv, 'class="dimensions"', {
            "<b>Current Dimensions:</b> ",
            D3(x), " × ", D3(y)
        }}
    }
end
```

Once the dialog has been created, its contents can be updated through Vectric's native `SetInnerHtml()` method:

```lua
Dlg:SetInnerHtml("details", DimensionSummary(X, Y))
```

The result is conceptually equivalent to:

```html
<div id="details">
    <div class="dimensions">
        <b>Current Dimensions:</b> 12.5 × 8
    </div>
</div>
```

Subsequent calls can replace the displayed values without changing the outer `details` container.

This example deliberately uses ordinary HTML text instead of Vectric-bound fields.

Because the HTML is generated afresh whenever the dimensions change, there is no need to establish native label bindings for the inner content.

Why not use AddElements here?

`AddElements()` appends content to Gadget Factory's internal collection during interface construction.

It does not return an HTML string suitable for passing to `SetInnerHtml()`.

`GetElementsHtml()`, by contrast, returns the generated markup directly.

This makes the latter more convenient for creating replacement content at runtime.

For additional discussion of interface updates, see Dynamic HTML (p. 364).

How GetElementsHtml works internally

The entire rendering mechanism is comparatively small.

The central implementation is:

```lua
GF.GetElementsHtml = function(elements)
    local html = {}

    for _, item in ipairs(elements) do
        if type(item) == "string" then
            html[#html + 1] = item

        elseif type(item) == "table" then
            if type(item[1]) == "function" then
                local render = assert(ElementRenderers[item[1]],
                    "No read-only HTML renderer for this element function")

                html[#html + 1] = render(table.unpack(item, 2))
            else
                html[#html + 1] = GF.GetElementsHtml(item)
            end
```

LUA / CONTINUED

```
        elseif type(item) == "function" then
            error("HTML element functions must be supplied in an element table", 2)
        end
    end

    return table.concat(html)
end
```

The basic traversal is nearly identical to that of `AddElements()`.

The critical difference is this statement:

LUA

```
local render = assert(ElementRenderers[item[1]],
    "No read-only HTML renderer for this element function")
```

Rather than executing the original function, Gadget Factory looks it up in the `ElementRenderers` table.

If a corresponding renderer exists, it is invoked with the remaining entries of the function-call table.

Its returned HTML is added to the local `html` collection.

After processing all entries, Gadget Factory concatenates the collected strings and returns the result.

Why use a renderer table?

A function in Lua is a first-class value and can serve as a table key.

Gadget Factory takes advantage of this by associating function references with other function references.

For example:

LUA

```
ElementRenderers[GF.AddDiv] = GF.GetDivHtml
```

This means that whenever `GetElementsHtml()` encounters a function-call table beginning with `GF.AddDiv`, it can retrieve and execute `GF.GetDivHtml()` instead.

There is no need to examine function names as strings, parse Lua source code, or implement separate syntax for read-only rendering.

The element's original arguments are preserved, but the operation performed with those arguments changes.

This is another example of Lua's flexible treatment of functions making a comparatively sophisticated facility possible with very little code.

Unsupported function entries

Unlike `AddElements()`, which can execute arbitrary functions supplied through function-call tables, `GetElementsHtml()` recognizes only functions registered in the private `ElementRenderers` table.

For example:

```lua
GF.GetElementsHtml {
    {GF.AddBinding, function(dlg)
        dlg:AddCheckBox("chkReady", true)
    end}
}
```

This raises an error because `GF.AddBinding()` has no registered HTML renderer.

Its purpose is to register a native binding, not generate HTML.

The same restriction applies to functions such as `GF.AddHandler()` and arbitrary programmer-defined functions.

A bare function is also invalid

This is not a valid element entry:

```lua
GF.GetElementsHtml {
    GF.AddDiv
}
```

The function must appear as the first entry of a function-call table:

```lua
GF.GetElementsHtml {
    {GF.AddDiv, {"<p>Hello</p>"}}
}
```

The renderer explicitly reports an error when it encounters a bare function.

Unsupported scalar values

Values that are neither strings nor tables are silently ignored, apart from the special error handling for bare functions.

For example:

```lua
GF.GetElementsHtml {
    "Value: ",
    42
}
```

Returns only:

```text
Value:
```

Numeric values must be converted to strings:

```lua
GF.GetElementsHtml {
    "Value: ",
    tostring(42)
}
```

This is consistent with the string-handling conventions of `AddElements()`.

Important considerations

- **HTML output:** `GetElementsHtml()` returns a string; it does not add content to Gadget Factory's dialog document.
- **Supported functions only:** Function-call tables must reference element functions with registered read-only renderers.
- **No native bindings:** Generating a field's HTML does not register its Vectric binding.
- **No event registration:** Rendering does not establish Gadget Factory callback handlers.
- **Nested tables:** Element collections are processed recursively.
- **Function-call syntax:** Functions must be supplied as the first entry of an element table, not as bare function values.
- **Array-style tables:** Entries are processed using `ipairs()`, so sequential integer indices are expected.
- **Optional arguments:** `false` placeholders may be used where necessary to preserve positional arguments in function-call tables.
- **String conversion:** Numbers and other non-string values are not automatically converted into HTML text.
- **No automatic separators:** Strings are concatenated without added spaces or line breaks. `GetDivHtml()` explicitly appends a newline after its closing tag.
- **No automatic escaping:** HTML strings and attribute values must be suitable for their destination. External data should be escaped before insertion.
- **Normal Lua evaluation:** Expressions used to build the element collection still execute normally, including any programmer-defined function calls.
- **Dynamic HTML:** Returned markup can be used with `SetInnerHtml()` or other appropriate HTML-update operations, but new Vectric field registrations are not created automatically.
- **Component rendering:** Gadget Factory's component libraries are not automatically given read-only renderers. This facility applies to the factory-owned element functions registered in `ElementRenderers`.

Related functions

- AddElement (p. 72) - Append raw HTML to the dialog.
- AddElements (p. 74) - Process element collections and execute their functions.

- AddDiv (p. 80) - Create an HTML container during dialog construction.
- AddControlLabel (p. 86) - Create an HTML label associated with a control.
- AddLabelSpan (p. 103) - Create a bound inline label.
- Field Primitives (p. 93) - Create native Vectric-bound fields.
- AddStateFlag (p. 340) - Create a hidden, Vectric-bound checkbox.
- AddBinding (p. 329) - Register native field-binding operations.
- Dynamic HTML (p. 364) - Update existing dialog HTML programmatically.
- InitializeScript (p. 34) - Construct the interface before dialog creation.

Summary

`GF.GetElementsHtml()` and `GF.GetDivHtml()` provide HTML-generation capabilities without executing the normal construction and binding operations of Gadget Factory's element primitives.

They preserve the familiar nested-table syntax while changing the result from an operation performed on the dialog's content collection into a reusable HTML string.

The mechanism relies on a private mapping of element functions to their read-only renderers, allowing the factory to reuse its established HTML construction logic without reproducing its side effects.

The essential distinction is:

`AddElements()` **builds the dialog.** `GetElementsHtml()` **builds a string.**

That distinction makes read-only rendering particularly useful for dynamic HTML generation, reusable markup, and specialized interfaces that do not require new native Vectric bindings.

REFERENCE 7.6

Dynamic HTML

Vectric's `HTML_Dialog` provides methods for modifying the HTML displayed in an open gadget dialog.

These methods allow programmers to replace individual elements, update entire sections of an interface, and construct controls dynamically without regenerating the complete HTML document.

Gadget Factory works naturally with this capability. Its HTML-generating functions can produce replacement markup while its existing event-dispatching and binding mechanisms continue to manage the Lua side of the interface.

However, dynamic HTML introduces several important considerations involving dialog initialization, element identifiers, native bindings, and Vectric's embedded browser engine.

This chapter documents both the supported mechanisms and practical findings from experiments conducted during Gadget Factory's development.

The two principal methods

Dynamic HTML manipulation is performed through methods of Vectric's native `HTML_Dialog` object.

Syntax

```lua
dlg:SetInnerHtml(id, html)
dlg:SetOuterHtml(id, html)
```

Parameters

Parameter	Description
`id`	Identifier of an existing HTML element to modify.
`html`	String containing the replacement HTML markup.

These are **Vectric methods, not Gadget Factory functions**.

They are called through the dialog object supplied to InitializeDialog (p. 49) or exposed as `GF.Dialog` after initialization.

SetInnerHtml

`SetInnerHtml()` replaces the contents of an existing HTML element while preserving the element itself.

Consider:

```HTML
<div id="details">
    <p>Original content</p>
</div>
```

Executing:

```LUA
Dlg:SetInnerHtml("details", "<b>Updated content</b>")
```

Produces the equivalent document structure:

```HTML
<div id="details">
    <b>Updated content</b>
</div>
```

The `details` container remains intact, retaining its ID, CSS classes, attributes, and position.

Only its contents are replaced.

SetOuterHtml

`SetOuterHtml()` replaces the specified element itself, including its opening and closing tags.

Given:

```HTML
<div id="details">
    <p>Original content</p>
</div>
```

Executing:

```LUA
Dlg:SetOuterHtml("details",
    '<section id="details"><b>Updated content</b></section>')
```

Produces:

```HTML
<section id="details">
    <b>Updated content</b>
</section>
```

The original `<div>` has been replaced by a `<section>`.

In this example, the replacement deliberately retains the ID `details`, allowing subsequent operations to locate the new element using the same identifier.

Choosing between the methods

Requirement	Preferred method
Replace the contents of an existing container	`SetInnerHtml`
Preserve a container's attributes and positioning	`SetInnerHtml`
Replace an element's tag or attributes	`SetOuterHtml`
Change an input element's HTML representation	`SetOuterHtml`
Replace an entire interface region	Either, depending on which elements must survive
Update the value of an existing Vectric-bound field	Native `Update...Field()` method

As a general rule, **prefer `SetInnerHtml()` when the containing element doesn't need to change**.

It preserves a stable point of reference in the document and reduces the possibility of accidentally removing elements needed by other operations.

Dialog initialization and the DOM

One of our earliest experiments uncovered an important distinction between creating an `HTML_Dialog` object and having a usable HTML document.

Gadget Factory's initialization sequence includes:

1. Executing InitializeScript (p. 34).
2. Generating the HTML document.
3. Creating the native `HTML_Dialog` object.
4. Registering native field bindings.
5. Executing InitializeDialog (p. 49).
6. Calling `ShowDialog()`.

The dialog object exists by step 3 and native bindings are applied by step 4. Those facts do not establish that DOM-dependent updates can be relied upon during `InitializeDialog()`.

An important experimental finding

During development, we found that calling `SetInnerHtml()` before `ShowDialog()` did not populate the intended element.

The same operation worked when invoked from a LuaButton callback after the document had loaded.

A native Aspire test also found that calling `GF.ExecutionEnabled(true)` from `InitializeDialog()` failed to enable the Execute button. That function uses `SetOuterHtml()`, so the timing concern applies to this DOM-dependent operation as well.

This means that code such as:

```lua
local function InitializeDialog(dlg)
    dlg:SetInnerHtml("details", "<b>Ready</b>")
end
```

should **not** be relied upon to initialize dynamically generated content.

The native dialog and its field bindings are available, but DOM modification cannot be relied upon at this point. These results do not conclusively prove that the DOM is absent; they establish the practical need to defer such updates.

The solution: post-load initialization

For interfaces that require HTML insertion immediately after opening, we successfully used a hidden LuaButton activated by the document's load event.

That callback executed after the browser had loaded the document, allowing dynamic HTML methods to operate normally.

The general sequence is:

```text
InitializeScript
       |
       v
Generate HTML
       |
       v
Create HTML_Dialog
       |
       v
Apply native bindings
       |
       v
Assign GF.Dialog and call InitializeDialog
       |
       v
ShowDialog
       |
       v
Document load event
       |
       v
Post-load LuaButton callback
       |
       v
SetInnerHtml / SetOuterHtml
```

A programmer may implement this explicitly, or use a Gadget Factory component that already manages its own post-load initialization.

For content that only needs updating in response to user interaction, an ordinary LuaButton callback is sufficient.

Practical rule: Construct and bind controls before displaying the dialog; perform dynamic HTML manipulation after its document has loaded.

Example: Updating a section of an interface

A common use of `SetInnerHtml()` is replacing an entire display section with freshly generated content.

Suppose a gadget maintains two numeric dimensions, `X` and `Y`.

During `InitializeScript()`, create a container and a button:

```lua
GF.AddDiv('id="divSummary"')

GF.AddElement([[
    <input type="button" class="LuaButton"
           id="btnRefresh" value="Refresh Summary">
]])
```

The container is intentionally empty.

Next, register a callback that generates the summary.

```lua
local D3 = GF.NumberFormatter(3)

GF.AddHandler("btnRefresh", function()
    local html = GF.GetElementsHtml {
        {GF.AddDiv, 'class="summary"', {
            "<b>Current Dimensions:</b> ",
            D3(X), " × ", D3(Y)
        }}
    }

    GF.Dialog:SetInnerHtml("divSummary", html)
end)
```

When the user clicks **Refresh Summary**, Gadget Factory constructs a new HTML string and passes it to Vectric.

The resulting content is equivalent to:

```html
<div id="divSummary">
    <div class="summary">
        <b>Current Dimensions:</b> 12.5 × 8
    </div>
</div>
```

Only the contents of `divSummary` are replaced.

The button remains unchanged, and the container retains its identity.

Why use GetElementsHtml?

The GetElementsHtml (p. 351) function returns an HTML string without registering native bindings.

That is exactly what this example needs.

Its contents are ordinary display markup, not Vectric-bound controls, so no additional field registrations are necessary.

This is also why `GF.AddElements()` would be inappropriate here. It appends content to Gadget Factory's initial document collection rather than returning a string for runtime insertion.

The importance of element identifiers

HTML identifiers serve several related purposes in Gadget Factory:

- Identifying elements for Vectric's native field methods.
- Locating elements for dynamic HTML replacement.
- Associating controls with event handlers.
- Providing targets for CSS selectors and JavaScript.

Our experiments established that **these identifiers play an especially important role when replacing existing controls**.

Replacing an existing bound control

Consider a manually constructed TextField.

During initialization:

```lua
GF.AddDiv('id="editHost"', {
    [[<input type="text" class="cssTXT" id="txtTest">]]
})

GF.AddBinding(function(dlg)
    dlg:AddTextField("txtTest", "Original")
end)
```

This creates an HTML input and registers the ID `txtTest` with Vectric.

Once the dialog is running, its HTML can be replaced:

```lua
Dlg:SetInnerHtml("editHost", [[
    <input type="text" class="cssTXT" id="txtTest">
]])
```

In our native Aspire experiments, a replacement TextField using the same ID remained accessible through the previously established Vectric binding.

Typing into the replacement control updated the value returned by the native field getter.

No second `AddTextField()` registration was required in that tested case.

What this establishes

Vectric's existing binding remained useful after the original HTML input was removed and a replacement with the same identifier was inserted.

The binding was therefore not permanently tied to the original HTML element instance.

This makes dynamic replacement considerably more flexible than we initially expected.

What it does not establish

Our tests do not prove that every possible native field type, browser event, or replacement structure behaves identically.

In particular, the observation concerns controls with an **existing binding and preserved ID**.

It does not establish that inserting an entirely new control ID automatically creates a native Vectric binding.

Nor does it establish that removing and recreating arbitrary interface structures will preserve every aspect of their previous state.

Preserving the ID is a tested and useful technique, not a universal guarantee for all conceivable controls.

Replacing HTML does not reinitialize field values

Another experiment exposed a subtler distinction.

We replaced an existing bound text input with markup containing an explicit value attribute:

```HTML
<input type="text" id="txtTest" value="Replacement">
```

The replacement did not display the specified value as expected.

However, typing into the field still updated the value accessible through the original Vectric binding.

This demonstrates that **replacing an element's HTML and changing its native field value are not equivalent operations**.

If a control already has a Vectric binding, use its corresponding native update method to change the bound value.

For example:

```LUA
Dlg:UpdateTextField("txtTest", "Replacement")
```

This is the intended API for updating a registered TextField's value, whereas `SetInnerHtml()` and `SetOuterHtml()` manipulate HTML markup.

The same distinction applies conceptually to other native field types.

Operation	Responsibility
`SetInnerHtml()`	Replace an element's contents
`SetOuterHtml()`	Replace an entire HTML element
`UpdateTextField()`	Update a bound text value
`UpdateDoubleField()`	Update a bound numeric value
`UpdateLabelField()`	Update a bound display label

Where a native update method is available and appropriate, prefer it over reconstructing the HTML field.

Dynamic HTML and LuaButton callbacks

Replacing HTML controls introduces another requirement: the replacement markup must preserve the mechanism that generates the relevant event.

An event handler can remain registered in Gadget Factory while the HTML needed to trigger it has disappeared.

LuaButton class requirements

Vectric recognizes callback controls through its special `LuaButton` HTML class.

For example:

```HTML
<input type="button" class="LuaButton"
       id="btnAction" value="Execute">
```

Our experiments established that dynamically inserted controls can generate LuaButton callbacks.

However, Vectric's callback recognition is particular about the class attribute.

Use `class="LuaButton"` exactly for callback elements.

Do not assume that adding additional CSS classes will preserve the callback behavior.

Instead, style LuaButton controls through their IDs or surrounding containers.

For example:

```CSS
#btnAction {
    width: 100px;
    height: 35px;
}
```

This avoids interfering with Vectric's callback recognition.

Preserving a callback during replacement

A callback registered through:

```lua
GF.AddHandler("btnAction", ProcessAction)
```

remains in Gadget Factory's dispatch table until explicitly changed or removed.

If the HTML button is replaced with another properly configured LuaButton using the same ID, the existing handler can continue to serve the replacement.

We successfully exercised this pattern during development.

There is no need to call `AddHandler()` again merely because the HTML representation changed.

In fact, attempting another registration using `AddHandler()` would raise a duplicate-handler error.

Use ReplaceHandler (p. 320) only if the associated Lua behavior itself needs to change.

A trap involving editable fields

Our experiments uncovered an additional requirement when dynamically replacing fields that use Gadget Factory's automatic change handling.

Consider an editable DoubleField constructed by the factory.

Its generated HTML includes an input whose `onchange` event activates a hidden LuaButton.

A simplified representation looks like this:

```html
<input type="text" class="cssDBL" id="txtDepth"
       onchange="document.getElementById('chg_txtDepth').click()">

<input type="button" class="LuaButton"
       id="chg_txtDepth" style="display:none;">
```

The visible input is not itself a LuaButton.

Instead, its `onchange` event activates the hidden control, which invokes Gadget Factory's registered callback.

What happened during testing

We replaced a previously bound input with a new element bearing the same field ID.

The native field binding remained operational, but the expected change callback did not execute.

The reason turned out to be straightforward: **the replacement markup had omitted the original change-handling mechanism**.

The field existed, and its value could be accessed, but its event bridge was missing.

The lesson

When replacing a control, preserve all the HTML required for its behavior.

For Gadget Factory's editable fields, that may include:

- The visible input and its original ID.
- The `onchange` attribute that activates the hidden button.
- The hidden LuaButton, if the replaced region also contained that element.
- The original handler registration in Gadget Factory's dispatcher.

If the hidden LuaButton remains outside the replaced region, it should not be duplicated.

The critical distinction is that **native field binding and event dispatching are independent mechanisms**.

Preserving one does not automatically recreate the other.

See Field Primitives (p. 93) and Event Handling (p. 320) for their respective implementations.

Replacing entire interface regions

Dynamic HTML replacement is not limited to individual controls.

During Gadget Factory's development, we successfully replaced the contents of larger HTML containers with generated collections of elements.

For example:

```lua
Dlg:SetInnerHtml("Components", table.concat(Content, "\n"))
```

This approach can update many related elements through a single operation.

Preserve stable containers

Suppose a dialog contains two independently managed regions:

```html
<div id="Components"></div>
<div id="DialogButtons"></div>
```

Both can be updated independently:

```lua
Dlg:SetInnerHtml("Components", componentHtml)
Dlg:SetInnerHtml("DialogButtons", buttonHtml)
```

This is often preferable to repeatedly replacing a shared parent.

If `DialogButtons` were nested inside `Components`, replacing the contents of `Components` would remove the existing button container unless the replacement HTML recreated it.

Our experiments confirmed that keeping independently updated host containers as siblings avoids this problem.

A useful design principle

Preserve the elements used as stable insertion points. Replace their contents rather than the insertion points themselves whenever practical.

This makes dynamic interfaces easier to maintain and reduces dependencies between separate regions.

The technique became particularly valuable as Gadget Factory's component system evolved.

Example: ExecutionEnabled

Gadget Factory itself uses `SetOuterHtml()` for a more specialized purpose.

The public `GF.ExecutionEnabled()` function controls whether the standard Execute button behaves as an active Vectric dialog button.

Its implementation switches between two HTML representations.

When enabling execution:

```lua
Dlg:SetOuterHtml("ButtonNotOK",
    [[<input type="button" id="ButtonOK"
        value="Execute">]])
```

When disabling execution:

```lua
Dlg:SetOuterHtml("ButtonOK",
    [[<input type="button" id="ButtonNotOK"
        value="Execute">]])
```

Notice that these replacements deliberately change the control's ID.

The enabled version uses the special `ButtonOK` identifier. The inactive version uses `ButtonNotOK`. Both omit `LuaButton` in the current implementation.

Raw `ButtonOK` and `ButtonCancel` buttons without the class worked in the Hello-Millie Aspire test (p. 289). Ordinary Lua callback controls retain their separate class and event-handling requirements.

This technique was tested successfully in Aspire and demonstrates that replacing an entire element can alter its functional role.

However, it should be regarded as a specialized, tested implementation rather than evidence that arbitrary ID changes preserve native bindings.

For normal gadget code, use the provided ExecutionEnabled (p. 289) function instead of recreating this mechanism manually.

Dynamic HTML versus native label updates

Both `SetInnerHtml()` and Vectric's native `UpdateLabelField()` can visibly change the contents of an existing HTML element.

For example:

```lua
Dlg:SetInnerHtml("labStatus", "<b>Complete</b>")
```

directly replaces the element's HTML contents.

By contrast:

```lua
Dlg:UpdateLabelField("labStatus", "Complete")
```

uses Vectric's registered LabelField mechanism.

Our experiments demonstrated that both approaches can produce visible updates.

They should nevertheless be treated as different operations.

`SetInnerHtml()` manipulates the HTML document directly. It does not require the target element to have been registered as a Vectric LabelField.

`UpdateLabelField()` operates through an existing native binding.

Visual similarity does not prove that the two methods have identical effects on Vectric's internal field state.

For ordinary bound labels, the native update method remains the clearer choice.

For arbitrary HTML containers and custom markup, `SetInnerHtml()` provides greater flexibility.

Dynamically inserted JavaScript

Dynamic HTML can contain markup that includes JavaScript event attributes or script elements.

However, inserting a `<script>` element into an existing document should not be assumed to execute that script automatically.

For reliable behavior, define required JavaScript in the original document and use dynamic HTML replacement primarily to insert or update markup.

For example, a static JavaScript function can operate on elements inserted later, provided their identifiers and expected structure are present when the function is invoked.

This distinction became important during Forge's development, where generated text needed to be staged and refreshed without relying on dynamically inserted script tags executing.

When generated content is source code or other arbitrary text, treat it as data. Escape it appropriately for the HTML or JavaScript context into which it is inserted.

Practical findings from our experiments

The following observations were established through development and native Aspire testing.

Observation	Result
`SetInnerHtml()` before `ShowDialog()`	Did not populate the intended content in our tests
`GF.ExecutionEnabled(true)` from `InitializeDialog()`	Failed to enable Execute in native Aspire verification; the operation uses `SetOuterHtml()`
`SetInnerHtml()` from a post-load callback	Worked
Dynamically inserted LuaButton controls	Generated callbacks when properly configured
LuaButton with additional CSS classes	Did not behave reliably in our tests
Replacing a bound TextField with the same ID	Preserved usable native field access in the tested case
Supplying an HTML `value` attribute during replacement	Did not initialize the previously bound field as expected
Replacing a field without its change-event bridge	Native field access remained, but the expected callback was lost
Replacing entire regions through stable containers	Worked
Replacing special Execute-button markup	Worked through the factory's tested ID-switching mechanism

These findings establish useful working conventions.

They are not a complete behavioral specification for every Vectric field type or every possible HTML replacement.

In particular, **newly introduced field IDs and arbitrary replacement structures should not be assumed to inherit existing native bindings**.

Important considerations

- **DOM update timing:** The native dialog and bindings exist during `InitializeDialog()`, but DOM-dependent updates are unreliable there. Perform them from a callback after the HTML document has loaded.
- **Stable IDs:** Preserve control identifiers when an existing native binding or callback is intended to remain in use.
- **No duplicate IDs:** Replacing a field with another element of the same ID requires removing the original element. Two simultaneous elements with the same ID can interfere with native field access.
- **Bindings are separate:** Injecting HTML does not automatically execute `GF.AddBinding()` or register new native fields.

- **Handler registrations persist:** Replacing markup does not automatically remove an existing Gadget Factory handler.
- **Event markup matters:** Callback controls must retain the necessary LuaButton class, event attributes, and hidden bridge controls.
- **Use native update methods:** When changing only the value of an existing bound field, prefer the corresponding `Update...Field()` method.
- **Preserve host containers:** Replacing a parent region can remove nested elements used as dynamic insertion points.
- **CSS remains applicable:** Replacement elements participate in the document's existing stylesheet rules.
- **Browser compatibility:** The embedded Trident engine may differ from modern browsers in its handling of HTML, CSS, and JavaScript.
- **Avoid modal dialogs during complex updates:** Our callback experiments exposed cases where opening a `MessageBox()` interfered with queued interface updates. Use in-dialog feedback where practical.
- **Read-only rendering:** `GF.GetElementsHtml()` generates markup without registering native fields or event handlers.
- **Test unfamiliar field types:** Do not assume that every native control behaves identically under dynamic replacement merely because TextFields and LuaButtons worked in the tested scenarios.

Choosing the appropriate technique

For ordinary interface development, the following guidelines cover most situations.

Task	Recommended technique
Construct the original interface	`GF.AddElements()`
Generate HTML without binding operations	`GF.GetElementsHtml()`
Update an existing TextField value	`Dlg:UpdateTextField()`
Update a bound label	`Dlg:UpdateLabelField()`
Replace a section's displayed contents	`Dlg:SetInnerHtml()`
Replace an element's entire HTML representation	`Dlg:SetOuterHtml()`
Change an existing callback's behavior	`GF.ReplaceHandler()`
Change appearance based on a Boolean condition	`GF.AddStateFlag()`
Enable or disable the standard Execute button	`GF.ExecutionEnabled()`

The simplest technique that accomplishes the task is generally the preferable one.

Dynamic HTML is powerful, but it should not replace ordinary native field updates or the framework's established primitives unnecessarily.

Related functions

- AddElement (p. 72) - Insert raw HTML during initial construction.
- AddElements (p. 74) - Construct HTML from nested Lua tables.
- AddDiv (p. 80) - Create stable HTML containers.
- GetElementsHtml (p. 351) - Generate replacement HTML without binding operations.
- AddBinding (p. 329) - Register native Vectric fields.
- Event Handling (p. 320) - Manage LuaButton callbacks.
- Field Primitives (p. 93) - Construct editable native fields.
- AddLabelSpan (p. 103) - Create bound inline labels.
- AddStateFlag (p. 340) - Control visual states through hidden checkboxes.
- ExecutionEnabled (p. 289) - Dynamically control the standard Execute button.
- InitializeDialog (p. 49) - Access the instantiated dialog before display.

Summary

Vectric's `SetInnerHtml()` and `SetOuterHtml()` methods allow gadgets to modify their interfaces while the dialog is running.

Gadget Factory complements these methods by providing reusable HTML generators, a centralized callback dispatcher, and native binding facilities.

The experimental findings establish three particularly valuable principles:

1. Timing matters. Dynamic HTML methods should be used after the browser's document is available, not merely after `HTML_Dialog` has been instantiated.

2. Identity matters. Preserving an existing control ID can allow its previously registered native binding and callback relationships to remain useful after HTML replacement.

3. Structure matters. Replacing markup must preserve any supporting elements, attributes, and event mechanisms required by the control.

When these principles are respected, dynamic HTML becomes a practical way to construct sophisticated interfaces without repeatedly recreating the entire dialog.

The underlying methods are simple. Their usefulness comes from understanding how they interact with Vectric's field bindings, Gadget Factory's event dispatcher, and the browser's document structure.

CHAPTER 08

08

Worked examples

Six complete gadgets, from a greeting to a layout grid.

IN THIS CHAPTER

Using the worked examples 380

Example 1: A First Dialog 382

Example 2: A Dimension Calculator 385

Example 3: Choices and Options 389

Example 4: Dependent Lists 392

Example 5: A Preview Workbench 396

Example 6: The Layout Grid 401

REFERENCE 8.1

Using the worked examples

A reference tells you what the pieces do. These examples put a few of those pieces to work together.

The six lessons progress from a single field through a small editing workbench to a gadget that creates vectors or native guide lines in the current job. Each is a complete, independent gadget with a copyable script, an accompanying Lua file, an explanation of the important decisions, and a short list of things to try.

Choose a starting point

Example	What you build	Principal techniques
1. A First Dialog (p. 382)	A name-and-greeting dialog	Lifecycle hooks, field callbacks, bound labels, Execute readiness
2. A Dimension Calculator (p. 385)	A rectangle area calculator	Numeric fields, ButtonGroup, validation, display precision
3. Choices and Options (p. 389)	A settings panel	RadioGroup, CheckboxGroup, defaults, StatusBar
4. Dependent Lists (p. 392)	A family-and-material selector	Combobox, ListBox, dynamic item collections, explicit selection handling
5. A Preview Workbench (p. 396)	An editable panel preview	ToolStrip, dynamic HTML, a separate model, undo, custom completion
6. The Layout Grid (p. 401)	A layout grid made from vectors or guides	Job geometry and origin, RadioGroup, LayerPicker, validation, native contours/guides, deferred creation

You can read them in order or start with the problem closest to the one you are solving. No example depends on a previous example's files or state.

Running the examples

These scripts use the framework-loading lines from Loading Gadget Factory (p. 9). They assume that Gadget Factory and its supporting libraries are already available in the documented working arrangement.

Copy an example's companion `.lua` file from the `Examples/Scripts` folder into a gadget folder in your working Vectric installation. The script must be beneath the `Vectric Files\Gadgets` root expected by those loading lines; running it directly from this documentation directory will not satisfy that requirement.

Launch it through Vectric's Gadgets menu, refreshing discovery as described in the Quick Start Guide (p. 5). Each lesson is a separate gadget; do not concatenate their scripts.

The first five examples use `Job = false` and report results without changing job geometry. Example 6 uses `Job = true` and creates real vectors on a chosen layer of the current sheet, or native guide lines, when you click Execute. It preserves existing vectors and guides; Cancel creates nothing. None of the examples needs templates, external data files, or toolpaths. The normal framework still generates and removes its temporary dialog HTML.

The complete scripts appear in the worked-example chapters. The companion Lua files contain the same code for convenient copying and testing.

A few conventions worth watching

The scripts construct their initial interface in `InitializeScript()`. Dynamic changes occur in callbacks after the document has loaded. They do not use `InitializeDialog()` to enable Execute or replace HTML.

In examples 1-4 and 6, Execute begins inactive. A field change, selection, or review action establishes whether the request is ready. Example 5 supplies its own Commit and Discard buttons and accepts any of its permitted model states.

Programmatic changes are followed by an explicit application refresh. The examples do not assume that selecting an item or restoring a checkbox automatically invokes the same callback as a user action.

Labels and application values are kept distinct where that distinction is useful. Numeric formatting is applied to the display without unnecessarily replacing the application's calculation values.

Verification

The scripts have been checked against the current factory and component source, parsed with Lua 5.2, and exercised with a simulated Vectric dialog host. The simulated checks cover initialization, callback behavior, validation, selection changes, undo/revert, and successful versus canceled completion. The Layout Grid also has simulated checks for native geometry calls, origins, units, sheet assignment, preservation of existing artwork, and failure handling. The guide extension also checks mode changes, picker disabling/restoration, guide positions and counts, and preservation of existing guides. Its API references and native exercise are listed in the lesson (p. 413).

Reconstructed dialog states have also been inspected in a modern browser for basic layout. Those previews use simulated field values; they do not exercise Vectric's event bridge or establish Trident rendering behavior.

The Layout Grid's earlier LayerPicker version passed native Aspire checks for existing/new layers, cancellation, active-layer preservation, and layer/Execute synchronization. On **23 October 2026**, Petey also confirmed native acceptance of `Enabled()` and the simplified Example 6: repeated mouse and keyboard mode changes, retained names, restored editing and suggestions, validity notifications, and synchronized status and Execute readiness. The example now uses `LP:Enabled(not guides)` instead of its own JavaScript. See the acceptance record and separate geometry, sheet, and side exercise (p. 414).

For the other examples, confirm field-change timing, the dependent lists, the workbench's custom completion buttons, and layout at your normal display scaling. A simulated host cannot establish those native rendering and event details.

Larger examples to explore

Sectio Divina connects a dimension editor to real gadget work. Its use of fields, grouped choices, custom actions, and completion hooks makes it a natural follow-on to these lessons. Look for `Sectio Divina.lua` when working through the larger example planned for the distribution.

Forge demonstrates how far the same foundation can go: a visual interface editor with a model, component properties, preview, saved designs, and Lua export. The preview workbench introduces part of that reasoning on a scale you can follow in one sitting.

REFERENCE 8.2

Example 1: A First Dialog

This first gadget has one job: ask for a name and greet the person who supplied it. That is enough to demonstrate the whole Gadget Factory lifecycle.

You construct the interface, respond to a field change, and process the result. Gadget Factory supplies the entry point, creates the dialog, binds the controls, and handles its completion.

What this example introduces

- A minimal `GF.GadgetDef`.
- `AddDiv()` and `AddTextField()` inside an element table.
- The difference between a control label and a bound display label.
- Enabling the standard Execute button from a user callback.

Try it

1. Open the gadget. Execute should initially be inactive.
2. Enter a name and press **Tab**. The prompt changes and Execute becomes available.
3. Replace the name with spaces and press Tab again. Execute becomes inactive.
4. Enter a name and click Execute. The dialog closes, then the greeting appears.
5. Run it again and click Cancel. There should be no greeting.

A TextField change is committed when you finish editing it; this example does not promise an update on every keystroke.

Complete script

Companion file: 01 - A First Dialog.lua (also attached to this PDF). See Running the examples (p. 380) before launching it.

```lua
-- VECTRIC LUA SCRIPT
require "strict"

local root = assert(debug.getinfo(1, "S").source:sub(2):gsub("/", "\\"):match("^(.-\\Vectric Files\\Gadgets\\)"), "Cannot locate Vectric Files\\Gadgets root.")
local GF = assert(loadfile(root .. "GadgetFactory.gfm"), "Cannot load GadgetFactory.")(root)

local Name = ""

local function Trim(text)
```

LUA / CONTINUED

```lua
    return (text:gsub("^%s+", ""):gsub("%s+$", ""))
end

local function NameChanged(value)
    Name = Trim(value)
    local ready = Name ~= ""
    GF.Dialog:UpdateLabelField("labPrompt",
        ready and "Ready. Execute will greet you."
              or "Please enter a name.")
    GF.ExecutionEnabled(ready)
end

local function InitializeScript()
    GF.AddDiv('id="intro" style="margin:20px;"', {
        "<h1>A first conversation</h1>",
        "<p>Enter your name, then press Tab to finish editing.</p>",
        {GF.AddControlLabel, "Name: ", "txtName"},
        {GF.AddTextField, "txtName", Name,
            false, false, 240, NameChanged},
        "<p>",
        {GF.AddLabelSpan, "labPrompt", "Please enter a name."},
        "</p>"
    })
end

local function ProcessResults()
    -- Read the final field value after successful dialog completion.
    Name = Trim(GF.Dialog:GetTextField("txtName"))
    if Name == "" then
        DisplayMessageBox("No name was supplied.")
        return
    end
    DisplayMessageBox("Hello, " .. Name .. "!")
end

GF.GadgetDef = {
    Title = "Example 1 - A First Dialog",
    Width = 570,
    Height = 330,
    Job = false,
    InitializeScript = InitializeScript,
    ProcessResults = ProcessResults,
}
```

Why it works

`InitializeScript()` describes the initial document. It does not need a dialog object.

The field's `NameChanged` callback runs later, when the user commits an edit. At that point it can update the bound prompt and call `GF.ExecutionEnabled()`. There is no need for an `InitializeDialog()` hook.

`AddControlLabel()` produces the label associated with the text input. `AddLabelSpan()` creates a separate display field that can be updated through `UpdateLabelField()`.

The `false` placeholders keep the element table's positional arguments intact while leaving the text field in ordinary document flow. Its containing div is named `intro`, so it does not opt into the shared stylesheet's `div` ID-prefix positioning convention.

`ProcessResults()` reads the final name only after successful completion. Its message box is outside the live dialog interaction. The small final check also prevents an empty greeting if the field's final value is unexpectedly blank.

Take it a little further

Replace the greeting with a summary of a proposed operation. Keep the actual operation in `ProcessResults()`; the field callback should prepare and validate the request.

Where to read more

- Gadget hooks (p. 11)
- Field change callbacks (p. 96)
- Element tables (p. 74)
- The standard Execute button (p. 289)

REFERENCE 8.3

Example 2: A Dimension Calculator

A useful dialog often needs to calculate something before it can do anything. This example calculates the area of one rectangular piece and the total area for several copies.

It is a unit-neutral calculator: use inches for both dimensions to obtain square inches, or millimetres for both to obtain square millimetres. It does not read a job's units or create any geometry.

What this example introduces

- DoubleFields and an IntegerField.
- A shared refresh routine called by several controls.
- ButtonGroup actions.
- Updating fields programmatically.
- Formatting results while retaining calculation values in Lua.

Try it

1. Click **Calculate** without editing anything. The initial values produce 100 square units per piece and 200 in total.
2. Change the dimensions or the number of copies, then leave the edited field.
3. Click **Swap X/Y**. The dimensions exchange places; the area stays the same.
4. Enter zero or a negative dimension. Execute should become inactive.
5. Restore positive dimensions and use a whole copy count from 1 to 1000. Execute reports the total after the dialog closes.

Vectric's native numeric fields handle their own numeric input rules. The example adds the application rule that dimensions must be positive and the copy count must be within its chosen range.

Complete script

Companion file: 02 - A Dimension Calculator.lua (also attached to this PDF). See Running the examples (p. 380) before launching it.

```lua
-- VECTRIC LUA SCRIPT
require "strict"

local root = assert(debug.getinfo(1, "S").source:sub(2):gsub("/", "\\"):match("^(.-\\Vectric Files\\Gadgets\\)"), "Cannot locate Vectric Files\\Gadgets root.")
local GF = assert(loadfile(root .. "GadgetFactory.gfm"), "Cannot load GadgetFactory.")(root)
```

LUA / CONTINUED

```lua
local Width, Height, Copies = 12.5, 8, 2
local D3 = GF.NumberFormatter(3)

local function Valid()
    return Width == Width and Height == Height
        and Width > 0 and Height > 0
        and Width < math.huge and Height < math.huge
        and Copies >= 1 and Copies <= 1000
        and Copies == math.floor(Copies)
        and Width * Height * Copies < math.huge
end

local function Refresh()
    local ready = Valid()
    local text = "Use positive dimensions and 1 to 1000 whole copies."
    if ready then
        text = "Area per piece: " .. D3(Width * Height)
            .. " | Total area: " .. D3(Width * Height * Copies)
            .. " square units"
    end
    GF.Dialog:UpdateLabelField("labArea", text)
    GF.ExecutionEnabled(ready)
end

local function WidthChanged(value)
    Width = value
    Refresh()
end

local function HeightChanged(value)
    Height = value
    Refresh()
end

local function CopiesChanged(value)
    Copies = value
    Refresh()
end

local function SwapDimensions()
    Width, Height = Height, Width
    GF.Dialog:UpdateDoubleField("txtWidth", tonumber(D3(Width)))
    GF.Dialog:UpdateDoubleField("txtHeight", tonumber(D3(Height)))
    Refresh()
end

local function InitializeScript()
    GF.LoadComponents {"ButtonGroup"}
    GF.AddStyle([[
        #calculator { margin:20px; }
        #calculator .entry { margin:12px 0; }
        #calculator label { display:inline-block; width:85px; }
        #labArea { position:absolute; left:20px; top:285px; width:560px; }
    ]])
    GF.AddDiv('id="calculator"', {
        "<h1>A little arithmetic</h1>",
        "<p>Use the same unit for both dimensions.</p>",
```

LUA / CONTINUED

```lua
        {GF.AddDiv, 'class="entry"', {
            {GF.AddControlLabel, "Width:", "txtWidth"},
            {GF.AddDoubleField, "txtWidth", Width,
                false, false, 110, WidthChanged}
        }},
        {GF.AddDiv, 'class="entry"', {
            {GF.AddControlLabel, "Height:", "txtHeight"},
            {GF.AddDoubleField, "txtHeight", Height,
                false, false, 110, HeightChanged}
        }},
        {GF.AddDiv, 'class="entry"', {
            {GF.AddControlLabel, "Copies:", "txtCopies"},
            {GF.AddIntegerField, "txtCopies", Copies,
                false, false, 110, CopiesChanged}
        }}
    })
    GF.ButtonGroup.New(1, 2):WithContent({
        {"Calculate", Refresh},
        {"Swap X/Y", SwapDimensions}
    }, 12, 225)
    GF.AddLabelField("labArea", "Click Calculate, or change a field.")
end

local function ProcessResults()
    if not Valid() then
        DisplayMessageBox("The dimensions or copy count are invalid.")
        return
    end
    DisplayMessageBox("Total area: "
        .. D3(Width * Height * Copies) .. " square units.")
end

GF.GadgetDef = {
    Title = "Example 2 - A Dimension Calculator",
    Width = 640,
    Height = 430,
    Job = false,
    InitializeScript = InitializeScript,
    ProcessResults = ProcessResults,
}
```

One calculation, several callers

`Refresh()` computes the displayed result and decides whether Execute is available. Field callbacks, Calculate, and Swap X/Y all use it.

Calculate is also a convenient way to review the valid initial values. It runs from a button callback after the document has loaded, so the example does not need to enable Execute during `InitializeDialog()`.

The final validation is repeated in `ProcessResults()`. In a working gadget, that is where the calculated dimensions would become an actual request to create or modify something.

Display precision and working precision

Gadget Factory formats DoubleFields to three decimal places. Its change callback receives the value read before that formatting, so the example retains the callback's value in `Width` or `Height`.

`D3()` formats the result for display. Swapping dimensions also formats the visible fields, while leaving the Lua values unchanged.

Editing a displayed number again can, of course, establish a new working value. Retaining precision is a deliberate choice about the application's model, not a promise that a rounded display can reconstruct discarded digits.

Programmatic updates need application updates

`UpdateDoubleField()` changes the displayed field. It does not take the place of the example's own assignment to `Width` and `Height`, nor does this script rely on it invoking the field callbacks. Swap X/Y updates the model, updates the controls, and then explicitly refreshes the calculation.

Where to read more

- DoubleField formatting and callbacks (p. 97)
- IntegerFields (p. 98)
- NumberFormatter (p. 281)
- ButtonGroup actions (p. 160)

REFERENCE 8.4

Example 3: Choices and Options

Some choices are alternatives; others are independent switches. A RadioGroup expresses the first kind, while a CheckboxGroup expresses the second.

This example builds a small settings panel for an imagined cutting operation. It reports the requested settings; it does not create toolpaths.

What this example introduces

- RadioGroup labels and associated values.
- CheckboxGroup keys, labels, and Boolean values.
- Reading multiple checkbox values.
- Restoring defaults without relying on callbacks.
- A StatusBar whose text comes from application state.

Try it

1. Click **Review** to accept the initial settings.
2. Change the cutting mode and toggle the additional options.
3. Notice that the summary shows the mode's associated value, such as `finish`, rather than its visible label.
4. Click **Defaults**. Roughing, a final pass, and no separate group should be restored.
5. Click Execute to see the final settings, or Cancel to discard the request.

Complete script

Companion file: 03 - Choices and Options.lua (also attached to this PDF). See Running the examples (p. 380) before launching it.

```lua
-- VECTRIC LUA SCRIPT
require "strict"

local root = assert(debug.getinfo(1, "S").source:sub(2):gsub("/", "\\"):match("^(.-\\Vectric Files\\Gadgets\\)"), "Cannot locate Vectric Files\\Gadgets root.")
local GF = assert(loadfile(root .. "GadgetFactory.gfm"), "Cannot load GadgetFactory.")(root)

local Mode, Options
local Status = "Choose some options, or click Review."
local Summary = "No settings reviewed yet."

local function ReadSummary()
```

LUA / CONTINUED

```lua
    local mode = Mode:Selected()
    local finalPass, separateGroup =
        Options:GetValues("FinalPass", "SeparateGroup")
    return "Mode: " .. tostring(mode)
        .. "; final pass: " .. (finalPass and "yes" or "no")
        .. "; separate group: " .. (separateGroup and "yes" or "no")
end

local function Refresh()
    Summary = ReadSummary()
    Status = "Settings ready."
    GF.Dialog:UpdateLabelField("labSummary", Summary)
    GF.StatusBar.Normal(true)
    GF.ExecutionEnabled(true)
end

local function RestoreDefaults()
    Mode:Select(1)
    Options:SetValues {FinalPass = true, SeparateGroup = false}
    -- Programmatic changes do not invoke the selection callbacks.
    Refresh()
end

local function InitializeScript()
    GF.LoadComponents {"RadioGroup", "CheckboxGroup", "ButtonGroup", "StatusBar"}
    GF.AddDiv('style="margin:20px;"', {
        "<h1>Choose the work, then its options</h1>"
    })
    Mode = GF.RadioGroup.New("Cutting mode", Refresh)
        :WithContent({
            {"Roughing", "rough"},
            {"Finishing", "finish"},
            {"Both", "both"}
        }, 20, 80, 210):Select(1)

    Options = GF.CheckboxGroup.New("Additional options", Refresh)
        :WithContent({
            {"FinalPass", "Include a final pass", true},
            {"SeparateGroup", "Use a separate group", false}
        }, 270, 80, 280)

    GF.ButtonGroup.New(1, 2):WithContent({
        {"Review", Refresh},
        {"Defaults", RestoreDefaults}
    }, 12, 235)

    GF.AddDiv('style="position:absolute;left:20px;top:300px;width:530px;"', {
        {GF.AddLabelSpan, "labSummary", Summary}
    })
    GF.StatusBar.Init {
        GetLeft = function() return Status end,
        GetRight = "Settings example"
    }
end

local function ProcessResults()
    DisplayMessageBox("Requested settings:\n" .. ReadSummary())
end
```

LUA / CONTINUED

```
GF.GadgetDef = {
    Title = "Example 3 - Choices and Options",
    Width = 620,
    Height = 500,
    Job = false,
    InitializeScript = InitializeScript,
    ProcessResults = ProcessResults,
}
```

Labels for people, values for the program

The RadioGroup presents “Finishing” but returns `"finish"`. Keeping a stable application value separate from the visible caption makes it easier to revise wording without rewriting the operation that consumes the choice.

CheckboxGroup adds a key because each checkbox represents a separate setting. Here, `FinalPass` identifies the setting, “Include a final pass” is its caption, and `true` is its initial value.

`GetValues("FinalPass", "SeparateGroup")` returns two values in the requested order. This lets the summary remain independent of the order in which Lua visits table entries.

Restoring defaults is an explicit operation

`Mode:Select(1)` and `Options:SetValues(...)` update the controls. They do not invoke the application's selection callbacks.

The Defaults action therefore finishes by calling `Refresh()` itself. That one explicit call keeps the summary, StatusBar, and Execute state in step.

A shared callback can ignore extra arguments

RadioGroup and CheckboxGroup provide different callback arguments. This example does not need them: it rereads the complete settings through the component methods. Lua permits `Refresh()` to ignore arguments it does not declare.

When a particular change matters, use the supplied arguments instead. A CheckboxGroup callback can accept `key, checked, group`; a RadioGroup callback can accept `value, group`.

Where to read more

- Reading the selected radio value (p. 125)
- Reading checkbox values (p. 141)
- Programmatic updates (p. 145)
- StatusBar getters (p. 299)

REFERENCE 8.5

Example 4: Dependent Lists

A short list can determine the contents of a longer one. Here, choosing a stock family fills a ListBox with the materials belonging to that family.

This is the same arrangement you might use for a category and its templates, a machine and its tools, or an operation and its available destinations. The example uses an ordinary Lua table, so no external files or job data are needed.

What this example introduces

- A Combobox with an explicit empty choice.
- A ListBox populated after the dialog opens.
- Replacing items with `SetItems()`.
- Clearing and selecting items programmatically.
- Keeping application readiness synchronized with the current selection.

Try it

1. Open the gadget. The family dropdown should be blank, the material list empty, and Execute inactive.
2. Choose **Sheet goods**, then choose **Birch plywood**. Execute becomes available.
3. Change the family to **Solid timber**. The old selection must clear and Execute must become inactive.
4. Click **First item** to select Oak programmatically.
5. Click **Clear**. The available materials stay visible, but no material remains selected.
6. Choose the blank family entry. The material list becomes empty.
7. Make a complete choice and click Execute. The result reports the stored keys, such as `sheet` and `birch`.

Complete script

Companion file: 04 - Dependent Lists.lua (also attached to this PDF). See Running the examples (p. 380) before launching it.

```lua
-- VECTRIC LUA SCRIPT
require "strict"

local root = assert(debug.getinfo(1, "S").source:sub(2):gsub("/", "\\"):match("^(.-\\Vectric Files\\Gadgets\\)"), "Cannot locate Vectric Files\\Gadgets root.")
local GF = assert(loadfile(root .. "GadgetFactory.gfm"), "Cannot load GadgetFactory.")(root)
```

LUA / CONTINUED

```lua
local Families, Materials
local Stock = {
    sheet = {
        {"MDF", "mdf"},
        {"Birch plywood", "birch"},
        {"Hardboard", "hardboard"}
    },
    solid = {
        {"Oak", "oak"},
        {"Maple", "maple"},
        {"Walnut", "walnut"}
    }
}

local function Refresh()
    local material, index = Materials:Selected()
    GF.Dialog:UpdateLabelField("labChoice",
        index > 0 and ("Selected material key: " .. material)
                  or "Select a material from the list.")
    GF.ExecutionEnabled(index > 0)
end

local function FamilyChanged(family)
    Materials:SetItems(Stock[family] or {})
    -- A new family requires a fresh choice, even if future lists overlap.
    Materials:Select(0)
    Refresh()
end

local function ChooseFirst()
    if Materials:Count() > 0 then
        Materials:Select(1)
    end
    Refresh()
end

local function ClearChoice()
    Materials:Select(0)
    Refresh()
end

local function InitializeScript()
    GF.LoadComponents {"Combobox", "ListBox", "ButtonGroup"}
    GF.AddDiv('style="margin:20px;"', {
        "<h1>One choice leads to another</h1>",
        "<p>Choose a stock family, then a material.</p>"
    })

    Families = GF.Combobox.New("Stock family", FamilyChanged)
        :AllowEmptySelection()
        :WithContent({
            {"Sheet goods", "sheet"},
            {"Solid timber", "solid"}
        }, 20, 105, 240):Select(0)

    Materials = GF.ListBox.New("Materials", Refresh)
        :WithContent({}, 300, 140, 260, 170)
```

LUA / CONTINUED

```lua
    GF.ButtonGroup.New(1, 2):WithContent({
        {"First item", ChooseFirst},
        {"Clear", ClearChoice}
    }, 12, 325)

    GF.AddLabelField("labChoice", "Choose a stock family.", 20, 390)
end

local function ProcessResults()
    local family, familyIndex = Families:Selected()
    local material, materialIndex = Materials:Selected()
    if familyIndex == 0 or materialIndex == 0 then
        DisplayMessageBox("No complete material choice was supplied.")
        return
    end
    DisplayMessageBox("Family key: " .. family .. "\nMaterial key: " .. material)
end

GF.GadgetDef = {
    Title = "Example 4 - Dependent Lists",
    Width = 640,
    Height = 530,
    Job = false,
    InitializeScript = InitializeScript,
    ProcessResults = ProcessResults,
}
```

Keep identity separate from position

The displayed label “Birch plywood” maps to the value `"birch"`. The value identifies the material; its one-based index only tells us where it currently appears.

This matters when lists are reordered or refreshed. An index is a position, not a durable description of the user's choice.

Decide what a new list means

ListBox normally tries to preserve a selection when the corresponding item survives a `SetItems()` update. Here the application deliberately adopts a stricter rule: changing the stock family always requires a fresh material choice.

That is why `FamilyChanged()` calls `Select(0)` after replacing the items. It then calls `Refresh()` to clear the summary and disable Execute.

Neither `SetItems()` nor a programmatic `Select()` should be treated as a substitute for the application's own selection callback. The First item and Clear buttons also refresh explicitly.

Let the components manage their event plumbing

Combobox and ListBox arrange their own native bindings, event bridges, and post-load initialization. The gadget uses their public methods instead of replacing their internal markup.

The application only changes the lists from user callbacks after the document has loaded. There is no need to put DOM-dependent work into `InitializeDialog()`.

Where to read more

- The optional blank dropdown entry (p. 197)
- Replacing the item collection (p. 225)
- Selected value and index (p. 223)
- ListBox document readiness (p. 239)
- Execute timing (p. 291)

REFERENCE 8.6

Example 5: A Preview Workbench

This example is a small editing workbench. It maintains a panel description in Lua and renders that description as HTML. A ToolStrip changes the panel's dimensions and color; a text field supplies its title.

Undo restores the previous edit. Revert restores the opening state. Commit reports the final description, while Discard closes the dialog without processing it.

There is no file save, geometry change, or toolpath operation here. The example concentrates on the part an editor must get right first: keeping its model, preview, and actions in agreement.

What this example introduces

- A ToolStrip with arguments passed to a shared action.
- A Lua model kept separate from the rendered HTML.
- Read-only rendering with `GetElementsHtml()`.
- Updating a stable preview container.
- Snapshot-based undo and revert.
- Custom Commit and Discard controls supplied by ButtonGroup.

Try it

1. Change the title, then press Tab. The preview title should change.
2. Try a title such as `Roughing < Finishing & Cleanup`. The characters should appear as text, not become HTML.
3. Use W+, W-, H+, and H- to change the panel. The dimensions stop at the limits in the script.
4. Use C to cycle through three colors.
5. Click **Undo** several times and watch the title, size, color, and history count return through earlier states.
6. Make more edits and click **Revert**. Then click Undo: the state immediately before Revert should return.
7. Click **Commit** to report the current model.
8. Run it again, make changes, and click **Discard**. No result message should appear.

Complete script

Companion file: 05 - A Preview Workbench.lua (also attached to this PDF). See Running the examples (p. 380) before launching it.

```lua
-- VECTRIC LUA SCRIPT
require "strict"

local root = assert(debug.getinfo(1, "S").source:sub(2):gsub("/", "\\"):match("^(.-\\Vectric Files\\Gadgets\\)"), "Cannot locate Vectric Files\\Gadgets root.")
local GF = assert(loadfile(root .. "GadgetFactory.gfm"), "Cannot load GadgetFactory.")(root)

local Original = {title = "Sample panel", width = 180, height = 100, tone = 1}
local Colors = {"#dceaf4", "#e0eddc", "#f1e1e8"}
local History = {}
local Status = "Change the panel with the tools on the left."

local function Copy(state)
    return {
        title = state.title, width = state.width,
        height = state.height, tone = state.tone
    }
end

local Model = Copy(Original)

local EscapeText = GF.EscapeHtml

local function PreviewHtml()
    return GF.GetElementsHtml {
        {GF.AddDiv, string.format(
            'class="previewCard" style="width:%dpx;height:%dpx;background-color:%s;"',
            Model.width, Model.height, Colors[Model.tone]), {
            "<h2>" .. EscapeText(Model.title) .. "</h2>",
            "<p>" .. Model.width .. " x " .. Model.height .. " pixels</p>"
        }}
    }
end

local function Refresh()
    GF.Dialog:SetInnerHtml("previewHost", PreviewHtml())
    GF.Dialog:UpdateTextField("txtTitle", Model.title)
    GF.StatusBar.Update()
end

local function Change(property, value)
    if Model[property] == value then return end
    History[#History + 1] = Copy(Model)
    Model[property] = value
    Status = "Preview updated."
    Refresh()
end

local function TitleChanged(value)
    Change("title", value)
end

local function Resize(args)
```

LUA / CONTINUED

```lua
    local value = Model[args.property] + args.step
    value = math.max(args.minimum, math.min(args.maximum, value))
    Change(args.property, value)
end

local function NextColor()
    Change("tone", Model.tone % #Colors + 1)
end

local function Undo()
    if #History == 0 then
        Status = "Nothing left to undo."
        GF.StatusBar.Update()
        return
    end
    Model = table.remove(History)
    Status = "Previous edit restored."
    Refresh()
end

local function Revert()
    History[#History + 1] = Copy(Model)
    Model = Copy(Original)
    Status = "Original panel restored. Undo can recover the previous panel."
    Refresh()
end

local function InitializeScript()
    GF.LoadComponents {"ToolStrip", "ButtonGroup", "StatusBar"}
    GF.NoDefaultButtons()
    GF.AddStyle([[
        #workbench { margin:20px 20px 0 66px; }
        #workbench p { margin:12px 0; }
        #previewHost {
            position:absolute; left:80px; top:145px;
            width:440px; height:280px; overflow:auto;
        }
        #previewHost .previewCard {
            box-sizing:border-box; padding:12px;
            border:1px solid #8093a3; border-radius:8px;
            overflow:auto; word-wrap:break-word;
        }
    ]])

    GF.ToolStrip.New(36, "left"):WithContent({
        {"W+", Resize, {property="width", step=20, minimum=140, maximum=380},
            "Make the panel wider"},
        {"W-", Resize, {property="width", step=-20, minimum=140, maximum=380},
            "Make the panel narrower"},
        {"H+", Resize, {property="height", step=20, minimum=80, maximum=240},
            "Make the panel taller"},
        {"H-", Resize, {property="height", step=-20, minimum=80, maximum=240},
            "Make the panel shorter"},
        {"C", NextColor, nil, "Cycle through the panel colors"}
    })

    GF.AddDiv('id="workbench"', {
        "<h1>A small preview workbench</h1>",
```

LUA / CONTINUED

```
        "<p>Edits change the preview. Commit reports your final design.</p>",
        {GF.AddControlLabel, "Title: ", "txtTitle"},
        {GF.AddTextField, "txtTitle", Model.title,
            false, false, 320, TitleChanged}
    })
    -- The initial preview is part of the original document.
    GF.AddDiv('id="previewHost"', {PreviewHtml()})

    GF.ButtonGroup.New(1, 4):WithContent({
        {"Undo", Undo},
        {"Revert", Revert},
        {"Commit", "ButtonOK"},
        {"Discard", "ButtonCancel"}
    }, 66, 440)

    GF.StatusBar.Init {
        GetLeft = function() return Status end,
        RightFormat = "Undo steps: %d",
        GetRight = function() return #History end
    }
end

local function ProcessResults()
    DisplayMessageBox("Committed preview:\n" .. Model.title
        .. "\n" .. Model.width .. " x " .. Model.height .. " pixels"
        .. "\nColor: " .. Colors[Model.tone])
end

GF.GadgetDef = {
    Title = "Example 5 - A Preview Workbench",
    Width = 650,
    Height = 570,
    Job = false,
    InitializeScript = InitializeScript,
    ProcessResults = ProcessResults,
}
```

The model is the thing being edited

`Model` describes the panel. The preview is a representation of that description.

`Change()` saves a copy of the previous state, changes one property, and refreshes the interface. Undo restores a saved copy. Revert is also recorded as an edit, so it can itself be undone.

The copy function is deliberately small because every value in this model is a string or a number. A model containing nested mutable tables would need an appropriate deeper copy; simply copying the outer table would leave shared objects in the history.

Render first, install only what needs behavior

`PreviewHtml()` uses `GetElementsHtml()` to return ordinary display markup. It does not install additional native fields or register new handlers every time the panel changes.

The original preview is included during `InitializeScript()`. Later refreshes replace only the contents of `previewHost`, leaving the host, title field, toolbar, and completion controls in place.

This example does not create interactive controls inside the preview. If you later preview a component, distinguish its preparation and rendering (p. 117) methods from installing a live component with `WithContent()`.

Text must stay text

The title comes from the user. Before it is placed between HTML tags, `GF.EscapeHtml()` converts characters that otherwise have markup meaning. In this script, `EscapeText` is a local alias for that canonical framework utility; it does not implement a separate escaping helper.

This is separate from updating the bound title field: `UpdateTextField()` receives the ordinary Lua string. Escaping belongs at the point where text is incorporated into HTML.

Give the toolbar its own space

A docked ToolStrip does not automatically reserve room in the rest of the document. The workbench content starts to its right, and the preview and action row use compatible coordinates.

The preview host's positioning is explicit. Its ID does not begin with `div`, so it does not accidentally acquire the stylesheet's ID-prefix positioning convention.

Custom completion is a different button arrangement

`NoDefaultButtons()` suppresses the standard buttons during construction. ButtonGroup supplies `ButtonOK` and `ButtonCancel` as string actions for Commit and Discard.

The other controls use function actions and the ordinary LuaButton callback mechanism. Completion controls use the native identifiers instead.

There is no call to `ExecutionEnabled()`: all permitted model states are acceptable, and this example supplies its own completion controls. `ProcessResults()` is the place to save or apply the model in a real application.

The raw completion-button mechanism and existing ButtonGroup behavior have been exercised in Aspire during framework development. This particular workbench layout still needs its own native check.

Beyond this example

Sectio Divina is a useful next study for applying interface state to a real gadget operation. Forge takes the model-and-preview idea much further, with component properties, selection, design persistence, and source export.

This small workbench does not reproduce either application. It gives you a compact place to understand some of the techniques before following them through a larger program.

Where to read more

- ToolStrip actions (p. 177)
- Read-only HTML generation (p. 351)
- Stable update containers (p. 373)
- Replacing the standard buttons (p. 293)
- Callback arguments (p. 320)

REFERENCE 8.7

Example 6: The Layout Grid

The first five examples work entirely inside their dialogs. This one leaves something useful in the job: a rectangular layout grid made from vectors on a chosen layer or from native guide lines.

Open a job, choose a margin and a number of rows and columns, and select **Vectors** or **Guides**. For vectors, choose an existing output layer or type a new name. Check the resulting cell dimensions, then Execute to create the chosen output. Cancel leaves the job alone.

The optional checkbox adds horizontal and vertical centerlines through the **overall grid**, not through every cell.

Undo and cleanup: Aspire's standard **Undo** does not reverse this gadget's changes. Remove guides through **View → Guide Lines → Delete All Guides**, or delete the generated vectors or their output layer. **Delete All Guides** removes existing guides too; delete an entire layer only when you want to remove everything it contains.

What this example introduces

- `Job = true` and the job object supplied to `InitializeScript()`.
- Material dimensions and origin-aware coordinates.
- Validation before enabling Execute, followed by a final check before creating geometry.
- RadioGroup for an output choice that changes validation and disables an irrelevant control.
- LayerPicker for an existing destination or a new layer name, with validity-transition callbacks.
- Read-only layer lookup followed by intentional layer creation.
- Native Contour and CadContour objects, or horizontal and vertical guide creation.
- Working on the current sheet without clearing existing artwork.

This example draws a flat, two-dimensional layout and rejects wrapped models. A double-sided job keeps its current side.

What will be drawn?

In **Vectors** mode, the grid consists of one closed outer boundary, one open vector for each internal vertical divider, and one open vector for each internal horizontal divider. Adjacent cells share their divider; the gadget does not draw a separate rectangle around every cell.

In **Guides** mode, each X position becomes a vertical guide and each Y position becomes a horizontal guide, including both outside positions on each axis. Native guides extend across the view; they are not trimmed to the grid rectangle. The margin identifies the outer guide positions, rather than clipping their length.

The margin applies equally to all four edges. Rows divide the material along Y; columns divide it along X.

With centerlines enabled, the gadget adds a line through the horizontal midpoint and another through the vertical midpoint. An existing divider already serves that purpose when its corresponding row or column count is even, so no duplicate is added within the new grid.

The same centerline choice applies to both output modes. These centerlines are reference marks. They do not change the requested row and column counts or the cell dimensions reported by StatusBar.

For vectors, the output name initially reads `GF Layout Grid`. Keep that familiar default, choose another existing layer from the suggestions, or type a new name. If the chosen layer already exists, the gadget appends to it. **Running the gadget again adds another grid**, including coincident vectors if the settings and destination are unchanged. It does not replace or clean up previous output.

Guides are created unlocked and made visible. **Repeated guide runs also append**, including coincident guides at unchanged positions. Earlier guides are left intact; use Aspire's native guide controls to remove any you no longer need.

Try it

1. Open a flat job. Launch the gadget and click **Check grid**. The StatusBar should report the calculated cell width and height in the job's units.
2. Enter a nonnegative margin and whole row and column counts from 1 to 100. Leave each field to commit its change.
3. Try a margin equal to half the material's smaller dimension. Execute should become inactive because no usable area remains.
4. Restore a valid margin and enable **Overall grid centerlines**. Try odd and even counts and observe the predicted count in each output mode.
5. Choose an existing, visible, unlocked vector layer from **Output layer**, click **Check grid**, then Execute. Inspect the new grid and confirm that its earlier artwork and the previously active layer are unchanged.
6. Run again and type a new output name. Click **Check grid**: the name should be accepted without creating a layer. Execute, then inspect the newly created layer and its grid.
7. Clear the output name or enter only spaces. Execute should become inactive. Type a usable name and check again. System layers are absent from the suggestions; entering an exact system-layer name is also rejected.
8. Try an ordinary locked, hidden, or bitmap layer. Click **Check grid** after changing between valid names: LayerPicker validates names, while the example performs the additional suitability checks.
9. Run again, type another unused name, change the settings, and Cancel. There should be no new vectors or layer. Repeating Execute instead must append another grid to the chosen destination.
10. In Vectors mode, repeat on another sheet, preferably one with different dimensions. Existing vectors on the first sheet must remain unchanged.
11. Try a centered origin and an offset origin. The margin should still be measured from the material edges.
12. Select **Guides**. Output layer should become inactive without losing its text, and the plan should report guides instead of vectors. Switch back: the picker should work again with the same destination.
13. Leave an invalid or unsuitable layer name in the picker, then select Guides. A valid grid should be executable because guides have no output layer. Switching back to Vectors must restore the layer checks.
14. Execute a guide grid. Verify the boundary positions, internal divisions, optional centerlines, visibility, and unlocked state. Confirm that no layer or vectors were created and that the active layer and selection remain unchanged.

15. Run Guides again and Cancel: no guides should be added and their visibility should stay as it was. Repeat with Execute to confirm that earlier guides are kept. Inspect guide behavior on another sheet and side as part of native testing.

The cap of 100 rows and 100 columns is a limit chosen for this teaching example, not a Gadget Factory or Vectric API restriction.

Complete script

Companion file: 06 - The Layout Grid.lua (also attached to this PDF). Follow Running the examples (p. 380) to place it in your working Vectric installation.

```lua
-- VECTRIC LUA SCRIPT
require "strict"

local root = assert(debug.getinfo(1, "S").source:sub(2):gsub("/", "\\"):match("^(.-\\Vectric Files\\Gadgets\\)"), "Cannot locate Vectric Files\\Gadgets root.")
local GF = assert(loadfile(root .. "GadgetFactory.gfm"), "Cannot load GadgetFactory.")(root)

local Job, Opening, Output, Options, LP
local MAX_COUNT = 100
local D3 = GF.NumberFormatter(3)
local Units, Status = "", "Click Check grid to calculate cell sizes."
local OutputScope = "Current sheet only"

local function Finite(value)
    return type(value) == "number" and value == value
        and math.abs(value) < math.huge
end

local function WrappedId(raw)
    local id = luaUUID()
    id:SetId(raw)
    return id
end

local function ContextUnchanged()
    return Job.Exists
        and WrappedId(Job.SheetManager.ActiveSheetId):IsEqual(Opening.sheet)
        and WrappedId(Job:GetActiveSideId()):IsEqual(Opening.side)
        and Job.MinX == Opening.x and Job.MinY == Opening.y
        and Job.Width == Opening.width and Job.Height == Opening.height
        and Job.InMM == Opening.inMM
end

local function LayerProblem(layer)
    if not layer then return nil end
    if layer.IsSystemLayer or layer.IsBitmapLayer then
        return "The output name belongs to a non-vector layer."
    end
    if layer.Locked then return "The output layer is locked." end
    if not layer.Visible then return "The output layer is hidden." end
end

local function AxisPositions(first, last, count)
```

LUA / CONTINUED

```lua
    local positions = {first}
    for i = 1, count do
        local value = i == count and last
            or first + (last - first) * (i / count)
        if value <= positions[#positions] or value > last then return nil end
        positions[#positions + 1] = value
    end
    return positions
end

local function MakePlan(margin, rows, columns, centerlines, output)
    if not ContextUnchanged() then
        return nil, "The sheet, side, units, or material bounds changed. Reopen the gadget."
    end
    if output ~= "Vectors" and output ~= "Guides" then
        return nil, "Choose Vectors or Guides for the output."
    end
    if not Finite(margin) or margin < 0 then
        return nil, "Margin must be a finite number greater than or equal to zero."
    end
    local function ValidCount(count)
        return Finite(count) and count == math.floor(count)
            and count >= 1 and count <= MAX_COUNT
    end
    if not ValidCount(rows) or not ValidCount(columns) then
        return nil, "Rows and columns must be whole numbers from 1 to 100."
    end
    if margin * 2 >= Job.Width or margin * 2 >= Job.Height then
        return nil, "The margin leaves no room for the grid."
    end

    local left, bottom = Job.MinX + margin, Job.MinY + margin
    local right = Job.MinX + Job.Width - margin
    local top = Job.MinY + Job.Height - margin
    local xs = AxisPositions(left, right, columns)
    local ys = AxisPositions(bottom, top, rows)
    if not xs or not ys then
        return nil, "The requested grid is too small at these coordinates."
    end

    local plan = {
        xs = xs, ys = ys, rows = rows, columns = columns, output = output,
        cellWidth = (right - left) / columns,
        cellHeight = (top - bottom) / rows,
        count = output == "Guides" and columns + rows + 2
            or 1 + (columns - 1) + (rows - 1)
    }
    -- With an even count, a normal divider already marks the center.
    if centerlines and columns % 2 == 1 then
        plan.centerX = left + (right - left) / 2
        local middle = (columns + 1) / 2
        if plan.centerX <= xs[middle] or plan.centerX >= xs[middle + 1] then
            return nil, "The vertical centerline cannot be represented accurately."
        end
        plan.count = plan.count + 1
    end
    if centerlines and rows % 2 == 1 then
        plan.centerY = bottom + (top - bottom) / 2
```

LUA / CONTINUED

```
        local middle = (rows + 1) / 2
        if plan.centerY <= ys[middle] or plan.centerY >= ys[middle + 1] then
            return nil, "The horizontal centerline cannot be represented accurately."
        end
        plan.count = plan.count + 1
    end

    if output == "Vectors" then
        if not LP:IsValid() then
            return nil, "Choose a nonblank output name that is not a system layer."
        end
        local layerName = LP:GetText()
        local problem = LayerProblem(Job.LayerManager:FindLayerWithName(layerName))
        if problem then return nil, problem end
        plan.layerName = layerName
    end
    return plan
end

local function ReadPlan()
    return MakePlan(
        GF.Dialog:GetDoubleField("txtMargin"),
        GF.Dialog:GetIntegerField("txtRows"),
        GF.Dialog:GetIntegerField("txtColumns"),
        Options:GetValues("Centerlines"),
        Output:Selected()
    )
end

local function CellDescription(plan)
    return "Cells: " .. D3(plan.cellWidth) .. " x "
        .. D3(plan.cellHeight) .. " " .. Units
end

local function Refresh()
    local guides = Output:Selected() == "Guides"
    LP:Enabled(not guides)
    OutputScope = guides and "Native guide lines" or "Current sheet only"
    GF.Dialog:UpdateLabelField("labOutputNote", guides
        and "Guides extend across the view. Existing guides are kept."
        or "Existing vectors on the chosen layer are kept.")
    local plan, problem = ReadPlan()
    if plan then
        Status = CellDescription(plan)
        GF.Dialog:UpdateLabelField("labPlan",
            plan.rows .. " rows x " .. plan.columns .. " columns; "
            .. plan.count .. (guides and " guides" or " vectors") .. " will be added.")
        GF.StatusBar.Normal(true)
    else
        Status = "Grid is not ready."
        GF.Dialog:UpdateLabelField("labPlan", problem)
        GF.StatusBar.Warning(true)
    end
    GF.ExecutionEnabled(plan ~= nil)
end

local function OutputChanged()
    Refresh()
```

LUA / CONTINUED

```
end

local function InitializeScript(job)
    Job = job
    if Job:IsWrappedModel() then
        DisplayMessageBox("This example requires a flat job rather than a wrapped model.")
        return false
    end
    if not Finite(Job.MinX) or not Finite(Job.MinY)
        or not Finite(Job.Width) or not Finite(Job.Height)
        or Job.Width <= 0 or Job.Height <= 0
        or not Finite(Job.MinX + Job.Width) or not Finite(Job.MinY + Job.Height) then
        DisplayMessageBox("The current material bounds are not usable.")
        return false
    end
    Opening = {
        sheet = WrappedId(Job.SheetManager.ActiveSheetId),
        side = WrappedId(Job:GetActiveSideId()),
        x = Job.MinX, y = Job.MinY, width = Job.Width, height = Job.Height,
        inMM = Job.InMM
    }
    Units = Job.InMM and "mm" or "in"
    local margin = math.min(Job.InMM and 6 or 0.25,
        math.min(Job.Width, Job.Height) / 10)

    GF.LoadComponents {"RadioGroup", "CheckboxGroup", "ButtonGroup", "StatusBar", "LayerPicker"}
    GF.AddStyle([[
        #gridSettings { margin:20px; }
        #gridSettings .entry { margin:12px 0; }
        #gridSettings label { display:inline-block; width:110px; }
        .cssRBG .cssRB { display:inline-block; width:130px; }
        .cssRBG .cssRB + .cssRB { margin-top:0; }
        #labPlan, #labOutputNote {
            position:absolute; left:20px; top:385px;
            width:650px; line-height:1.4;
        }
        #labOutputNote { top:440px; }
    ]])
    GF.AddDiv('id="gridSettings"', {
        "<h1>The Layout Grid</h1>",
        "<p>Material: " .. D3(Job.Width) .. " x " .. D3(Job.Height) .. " " .. Units
            .. "<br>Lower-left corner: (" .. D3(Job.MinX) .. ", "
            .. D3(Job.MinY) .. ") " .. Units .. "</p>",
        {GF.AddDiv, 'class="entry"', {
            {GF.AddControlLabel, "Margin (" .. Units .. "):", "txtMargin"},
            {GF.AddDoubleField, "txtMargin", margin,
                false, false, 110, Refresh}
        }},
        {GF.AddDiv, 'class="entry"', {
            {GF.AddControlLabel, "Rows:", "txtRows"},
            {GF.AddIntegerField, "txtRows", 4,
                false, false, 110, Refresh}
        }},
        {GF.AddDiv, 'class="entry"', {
            {GF.AddControlLabel, "Columns:", "txtColumns"},
            {GF.AddIntegerField, "txtColumns", 4,
                false, false, 110, Refresh}
        }}
```

LUA / CONTINUED

```
    })
    Output = GF.RadioGroup.New("Output", OutputChanged)
        :WithContent({{"Vectors", "Vectors"}, {"Guides", "Guides"}}, 330, 125, 330)
        :Select(1)
    Options = GF.CheckboxGroup.New("Grid options", Refresh)
        :WithContent({
            {"Centerlines", "Overall grid centerlines", false}
        }, 330, 210, 330)
    LP = GF.LayerPicker.New("Output layer", {
        Text="GF Layout Grid", OnValidityChanged=Refresh
    }):WithContent(330, 295, 330)
    GF.ButtonGroup.New(1, 1):WithContent({
        {"Check grid", Refresh}
    }, 12, 280)
    GF.AddLabelField("labPlan", "Check the grid before executing.")
    GF.AddLabelField("labOutputNote", "Existing vectors on the chosen layer are kept.")
    GF.StatusBar.Init {
        GetLeft = function() return Status end,
        GetRight = function() return OutputScope end
    }
end

local function BuildObjects(plan)
    local objects = {}
    local function AddContour(points)
        local contour = Contour(0.0)
        assert(contour:AppendPoint(points[1][1], points[1][2]),
            "Could not start a contour.")
        for i = 2, #points do
            assert(contour:LineTo(points[i][1], points[i][2]),
                "Could not append a line to a contour.")
        end
        objects[#objects + 1] = assert(CreateCadContour(contour),
            "Could not create a CAD contour.")
    end

    local left, right = plan.xs[1], plan.xs[#plan.xs]
    local bottom, top = plan.ys[1], plan.ys[#plan.ys]
    AddContour {{left, bottom}, {right, bottom}, {right, top},
        {left, top}, {left, bottom}}
    for i = 2, #plan.xs - 1 do
        AddContour {{plan.xs[i], bottom}, {plan.xs[i], top}}
    end
    for i = 2, #plan.ys - 1 do
        AddContour {{left, plan.ys[i]}, {right, plan.ys[i]}}
    end
    if plan.centerX then
        AddContour {{plan.centerX, bottom}, {plan.centerX, top}}
    end
    if plan.centerY then
        AddContour {{left, plan.centerY}, {right, plan.centerY}}
    end
    return objects
end

local function CreateGuides(plan)
    -- Native guides span the view; the four boundary positions need four guides.
    for _, x in ipairs(plan.xs) do Job:CreateVerticalGuide(x, false) end
```

LUA / CONTINUED

```lua
    for _, y in ipairs(plan.ys) do Job:CreateHorizontalGuide(y, false) end
    if plan.centerX then Job:CreateVerticalGuide(plan.centerX, false) end
    if plan.centerY then Job:CreateHorizontalGuide(plan.centerY, false) end
    Job:SetGuidesVisible(true)
end

local function ProcessResults()
    -- Read and validate the final bound values before any job mutation.
    local plan, problem = ReadPlan()
    if not plan then DisplayMessageBox(problem); return end

    if plan.output == "Guides" then
        local ok, err = pcall(CreateGuides, plan)
        Job:Refresh2DView()
        if not ok then
            DisplayMessageBox("The grid could not be completed.\n" .. tostring(err)
                .. "\nSome new guides may remain. Existing guides were not removed or replaced.")
            return
        end
        DisplayMessageBox("Added " .. plan.count .. " guides.\n" .. CellDescription(plan))
        return
    end

    -- Construct every object off-layer before starting the append operation.
    local built, objects = pcall(BuildObjects, plan)
    if not built then DisplayMessageBox(tostring(objects)); return end

    local manager = Job.LayerManager
    local previousLayer = manager:GetActiveLayer()
    local ok, err = pcall(function()
        local layer = manager:FindLayerWithName(plan.layerName)
            or manager:GetLayerWithName(plan.layerName)
        assert(layer, "Could not create the output layer.")
        local layerProblem = LayerProblem(layer)
        assert(not layerProblem, layerProblem)
        for _, object in ipairs(objects) do
            -- This Boolean selects the current sheet; it is not a selection flag.
            layer:AddObject(object, true)
        end
    end)

    if previousLayer then manager:SetActiveLayer(previousLayer) end
    Job:Refresh2DView()
    if not ok then
        DisplayMessageBox("The grid could not be completed.\n" .. tostring(err)
            .. "\nSome new vectors may remain on " .. plan.layerName
            .. ". Existing vectors were not removed or replaced.")
        return
    end
    DisplayMessageBox("Added " .. #objects .. " vectors to " .. plan.layerName
        .. ".\n" .. CellDescription(plan))
end

GF.GadgetDef = {
    Title = "Example 6 - The Layout Grid",
    Width = 730,
    Height = 640,
    Job = true,
```

LUA / CONTINUED

```
    InitializeScript = InitializeScript,
    ProcessResults = ProcessResults,
}
```

Requiring a job

Setting `Job = true` asks Gadget Factory to obtain a Vectric job and check that one exists before calling the initializer. The `job` argument passed to `InitializeScript()` is that validated object.

The example keeps the object in `Job`. It does not create a second job, open a file, or depend on the user's current selection.

The opening context records the current sheet and side, material bounds, and units. A final check refuses to proceed if those change. The script never switches sheets or sides to recover the old context; reopening the gadget is the clearer response to a changed job.

Coordinates follow the material

The lower-left corner of the material is not necessarily `(0, 0)`. Vectric exposes that corner through `Job.MinX` and `Job.MinY`.

For margin `m`, the usable bounds are:

TEXT

```
left   = Job.MinX + m
right  = Job.MinX + Job.Width - m
bottom = Job.MinY + m
top    = Job.MinY + Job.Height - m

cell width  = (right - left) / columns
cell height = (top - bottom) / rows
```

There is no need to reinterpret Vectric's corner or center-origin setting. The reported bounds already provide the coordinates the drawing needs, including an origin offset.

For example, take a 24 by 18 inch job with a centered origin. Its lower-left corner is `(-12, -9)`. A one-inch margin produces bounds from `(-11, -8)` to `(11, 8)`.

With three rows and four columns, each cell is 5.5 inches wide and 16/3 inches high. StatusBar displays the height as 5.333 inches, while the geometry calculation retains the unrounded value.

Each divider position is calculated directly from the span and its fractional position. Repeatedly adding a rounded cell size would accumulate error. The final endpoint is pinned to the far boundary, and the code rejects positions that collapse together at the available numerical precision.

Validation and displayed precision

`ReadPlan()` reads the bound controls for both interactive checking and final processing. The margin therefore uses the value currently held by the DoubleField, including Gadget Factory's normal three-decimal input formatting.

Subsequent cell calculations are not rounded. `D3()` is used when displaying the result.

The plan rejects negative or non-finite margins, invalid counts, and margins that leave no area. A zero margin is valid and places the boundary on the material edges. It also validates the selected output mode. For Vectors, it checks `LP:IsValid()` and the existing output layer without creating one, then captures `LP:GetText()` as `plan.layerName`. Guides bypass the layer-name and layer-suitability checks entirely; a disabled picker cannot prevent unrelated output.

In a three-row, four-column grid, the boundary and dividers make six vectors. Enabling centerlines adds the missing horizontal centerline, for seven vectors; the middle vertical divider already marks the other centerline. A four-by-four grid needs no additional centerlines.

The guide count is three greater than the vector count for equivalent settings: the single closed boundary vector is replaced by four separate guides. A three-row, four-column grid therefore needs nine guides, or ten with the missing horizontal centerline. Both modes use the same unrounded coordinate arrays.

Choosing the output type

RadioGroup (p. 121) associates the displayed choices with the values `"Vectors"` and `"Guides"`. Vectors is selected initially. Its Lua callback, `OutputChanged()`, reruns the existing plan checker so the output count, explanatory note, StatusBar, and Execute readiness follow the selected mode.

The same refresh function controls whether the output-layer picker is available:

```lua
local guides = Output:Selected() == "Guides"
LP:Enabled(not guides)
```

Enabled() (p. 253) belongs to the picker, so the example does not need JavaScript or knowledge of its internal controls. Guides disable the picker; Vectors enable it. Repeating the current state does nothing. The entered name is retained, and reenabling does not move focus into the picker.

DataPicker handles both the text field and suggestion arrow, their appearance, and closing any open suggestions. Changing availability does not change text validity or refresh the layer collection. The plan checker decides when the layer is relevant: Guides bypass layer checks; Vectors require a usable destination.

The final plan reads `Output:Selected()` again. It does not trust a remembered callback value or the picker's disabled appearance. A programmatic `Output:Select()` does not invoke the selection callback; call `Refresh()` afterward to synchronize the picker, explanatory text, StatusBar, and Execute readiness.

Choosing the output layer

LayerPicker (p. 252) combines an editable name with suggestions from the current job. `Text="GF Layout Grid"` establishes the familiar starting name without creating that layer. Choosing a suggestion or typing a new name still leaves the job unchanged.

`LP:IsValid()` rejects empty or whitespace-only input and exact current system-layer names. It preserves case and spaces and permits names that do not exist yet. LayerPicker's nonblank rule is a component policy; it is not a claim that Vectric itself prohibits an empty name.

The example has stricter requirements for its destination. `LayerProblem()` continues to reject system, bitmap, locked, and hidden layers. These checks belong to the grid operation, rather than to a generic layer-name picker.

`OnValidityChanged=Refresh` routes validity transitions through the existing plan checker. Layer validity affects Execute only in Vectors mode. Typing the first usable character, clearing the name, or entering a prohibited system name can update Execute readiness immediately. There is no initialization callback and no callback for a change between two valid names.

That last point matters: changing from a usable layer to a locked ordinary layer can leave the previous readiness indication unchanged. **Check grid** deliberately remains available to recheck all conditions, and `ProcessResults()` always validates the final bound values before creating geometry. A stale readiness indication cannot bypass those final checks.

Click **Check grid** when the dialog first opens, or select an output mode to run the same check. Although `IsValid()` can be queried in `InitializeDialog()`, the existing native timing restriction on `GF.ExecutionEnabled()` means this example keeps its initial button update in a loaded-dialog callback. It adds no new initialization event mechanism.

This example does not programmatically change the picker text. If you extend it to call `GF.Dialog:UpdateTextField(LP:Id("text"), name)`, follow that with `LP:RefreshLayers()` when notification is needed. A valid-to-valid assignment still will not call `Refresh()`, so invoke the full plan check explicitly if your extension needs an immediate suitability update. See the native update contract (p. 255).

Layers and sheets have different responsibilities

This section applies to Vectors mode. Guides never enter the layer lookup, creation, activation, or object-insertion path.

A layer identifies where the vectors are organized. It is not a substitute for their sheet assignment.

The SDK documents the second argument of `CadLayer:AddObject()` as `on_current_sheet`. Passing `true` places each newly created object on the current sheet. **It is not a request to select the object.**

The gadget does not enumerate, remove, transform, or reassign any existing object. That includes existing content on the output layer and content belonging to another sheet.

`FindLayerWithName()` is the read-only lookup. `GetLayerWithName()` creates a layer if it is missing, so it appears only in `ProcessResults()`, after the final validation and object construction. Both use the validated `plan.layerName`.

An existing output layer must be visible, unlocked, and suitable for vectors. The gadget reports an unsuitable layer rather than changing its properties. It does not recolor or rename an existing layer, and it restores the previously active layer after insertion. It does not clear or add to the selection.

The example deliberately does not call `LP:ActivateLayer()`. Direct insertion into a `CadLayer` does not require making that layer active. Keeping the existing lookup and insertion path preserves the user's active-layer state, including restoration if creating a new layer changes it.

Contours become job objects in two stages

A `Contour` contains the geometry. `AppendPoint()` sets its starting point, and `LineTo()` appends each straight segment.

Returning the boundary's final point to its starting point closes that contour. The dividers and centerlines contain one segment each and remain open.

`CreateCadContour()` wraps a completed Contour in the CAD object that a layer can own. Only `layer:AddObject(object, true)` inserts that object into the job.

The example checks the documented Boolean results of `AppendPoint()` and `LineTo()`. It does not invent a Boolean success return for `AddObject()`, whose SDK entry does not specify one.

Creating native guides

`CreateGuides(plan)` passes every X position to `Job:CreateVerticalGuide(x, false)` and every Y position to `Job:CreateHorizontalGuide(y, false)`. It adds the optional center positions only when an ordinary divider does not already occupy them. The `false` argument requests an unlocked guide.

After creation, `Job:SetGuidesVisible(true)` makes the result visible, and `Job:Refresh2DView()` refreshes the drawing. These calls happen only after Execute. Merely selecting Guides, checking the plan, or canceling does not create guides or change their visibility.

The SDK documents these creation methods and the guide visibility methods, but does not document guide enumeration, deletion, moving, rotation, or changing an existing guide's lock state. The example therefore appends its guides and leaves management of earlier guides to Aspire's native interface. The creation methods have no documented Boolean return contract; the example does not invent one.

Unlike `CadLayer:AddObject(object, true)`, the guide creation signatures have no sheet argument. The example keeps the active sheet and side unchanged and checks the opening context before creating either output. Exact guide scope across sheets and sides remains part of the native exercise; the StatusBar says **Native guide lines** in this mode instead of promising current-sheet vector behavior.

What transactional behavior means here

Nothing is added to the job while the dialog is open. The callbacks calculate a plan, report it, and decide whether Execute should be available.

After successful dialog completion, `ProcessResults()` rereads and validates the final settings. For Vectors, it constructs all the CAD objects off-layer before creating or modifying the output layer. A construction failure therefore leaves the job without any new grid objects.

The vector insertion stage appends the prepared objects. This is deferred execution, **not a claim that the SDK provides an atomic transaction**. If a native insertion raises an error partway through, some new vectors may remain. The script reports that possibility and does not attempt a rollback that might remove existing artwork.

Guides are created directly through the job API after validation; they have no equivalent off-layer staging step. A catchable error can likewise leave some new guides behind. The example reports partial completion without deleting earlier guides. Neither output mode claims all-or-nothing native insertion.

Vectric API verification

The native calls were checked against the local **Vectric Lua Interface for Gadgets** reference supplied with `Vectric_Gadget_SDK_V12`. Page numbers below are the printed SDK page numbers.

API	What the example relies on	SDK page
`VectricJob.Exists`, `InMM`, `MinX`, `MinY`, `Width`, `Height`	Job existence, units, material bounds	46-47
`VectricJob:IsWrappedModel()`, `Refresh2DView()`	Reject a wrapped model; refresh after insertion	49
`VectricJob:GetActiveSideId()`	Record and recheck the active side	50
`VectricJob:CreateHorizontalGuide(y, locked)`, `CreateVerticalGuide(x, locked)`	Create native guides at the calculated positions, unlocked	47
`VectricJob:SetGuidesVisible(visible)`	Show the guides after creation	49
`CadLayerManager:FindLayerWithName(name)`	Return an existing layer or nil without creating it	51
`CadLayerManager:GetLayerWithName(name)`	Return the layer, creating it if absent	51
`CadLayerManager:GetActiveLayer()`, `SetActiveLayer(layer)`	Preserve the active layer	51-52
`CadLayer.Locked`, `Visible`, `IsSystemLayer`, `IsBitmapLayer`	Check whether the existing layer is suitable	54
`CadLayer:AddObject(object, true)`	Transfer the new object to the layer on the current sheet	54
`CadSheetManager.ActiveSheetId`	Record and recheck the current sheet without changing it	57
`luaUUID()`, `SetId(raw_id)`, `IsEqual(id)`	Compare saved and current sheet/side identifiers	71
`Contour(0.0)`	Create an empty contour using the default tolerance	80

API	What the example relies on	SDK page
`Contour:AppendPoint(x, y)`, `LineTo(x, y)`	Build the boundary and straight dividers	82, 85
`CreateCadContour(contour)`	Create the layer-ownable CAD object	41, 64

The SDK's `Display_Job_Dimensions` and `Sheet_Example` samples also use `MinX` and `MinY` with the job's dimensions when locating material boundaries.

Verification and native exercise

Lua 5.2 parsing and simulated job/dialog checks cover missing jobs, corner/center/offset origins, inch and millimetre units, odd/even centerline counts, invalid input, current-sheet insertion, existing artwork, cancellation, changed context, and failures during construction or insertion. The LayerPicker integration also checks chosen existing and new destinations, blank and system names, unsuitable ordinary layers, validity callbacks, valid-to-valid changes, final name capture, and repeated execution without interaction-time mutation.

After replacing the example's JavaScript with `LP:Enabled(not guides)`, the combined regression harness passes **100 simulated scenarios with 39,296 assertions**. It checks each mode change against the picker's enabled state and permits only an internal state update; the text field and suggestions are preserved. The earlier vector and guide geometry checks remain in the suite.

A reconstructed Edge preview passes **95 interaction/layout checks**, including repeated mouse, arrow-key, and Space selection; retained text and field nodes; popup closure and reopening; and continued typing and validity callbacks. Its simulated native callbacks apply state patches produced by the actual Lua method. This checks the browser behavior, not Aspire's event bridge.

The component checks separately pass **104 Lua assertions**, **128 Edge assertions**, and **128 standalone Trident 11 assertions** for `Enabled()`. A further **20 Trident native COM replacements** exercise repeated disabling and reenabling. Existing LayerPicker and validity/focus checks also pass. These checks cover pending initialization, read-only queries, idempotent setters, instance isolation, disabling while focused, retained text, suggestion use, and callback behavior.

Native Aspire acceptance - 23 October 2026: Petey confirmed that all requested tests passed for `DataPicker:Enabled()`, its LayerPicker delegate, and this simplified example using `LP:Enabled(not guides)` in place of the example-level JavaScript.

The picker check gadget passed initial-state synchronization, Boolean state queries, repeated setters and toggles, instance independence, text retention, and disabling from a validity callback while focused with suggestions open. Reenabling restored editing and suggestion selection; state changes and queries did not add validity notifications.

Example 6 passed repeated Vectors/Guides switching by mouse and keyboard, including with suggestions open. The entered name survived, both picker controls disabled and reenabled together, typing and validity notifications continued, and StatusBar, Check grid, and Execute stayed synchronized. A blank layer name blocked Vectors while permitting Guides when the grid settings were valid. This records native acceptance of the picker API and simplified mode handling, in addition to the simulated and browser evidence above.

The native exercise should include two sheets with different dimensions, a chosen existing populated output layer, a newly named destination, a nonempty selection, and a centered or offset origin. On a double-sided job, check the active side as well. Verify both output modes' positions and counts, preservation of earlier artwork and guides, active layer, and selection, and that Cancel creates nothing. Check picker disabling and restoration in Aspire, including opening suggestions before changing output mode. For Guides, verify visibility, the unlocked state, repeated runs, and behavior across sheets and sides.

Where to read more

- Requiring a Vectric job (p. 11)
- The job passed to InitializeScript (p. 34)
- Processing after Execute (p. 25)
- Numeric field formatting (p. 97)
- Displaying calculated dimensions (p. 300)
- Reading the current output choice (p. 125)
- Enabling and disabling the output-layer picker (p. 253)
- Layer-name validation and its callback boundary (p. 254)
- Display and working precision (p. 388)

Index

Alphabetical topic and method index for the Gadget Factory Programmer's Reference.

Unqualified factory function names, such as `AddBinding()`, are called through `GF`. Component names identify their class: use the returned instance for colon methods, such as `group:Select()`. `StatusBar` uses dot calls through `GF.StatusBar`. `Core` services are for component authors. Native Vectric methods are marked `HTML_Dialog`.

For a reading sequence, see Table of Contents (p. ii).

A

AddBinding() (p. 329)
AddControlLabel() (p. 86)
AddDiv() (p. 80)
AddDoubleField() (p. 97)
AddElement() (p. 72)
AddElements() (p. 74)
AddHandler() (p. 320)
AddIntegerField() (p. 98)
AddLabelField() (p. 99)
AddLabelSpan() (p. 103)
AddStateFlag() (p. 340)
AddStyle() (p. 108)
AddTextField() (p. 98)
Advanced topics (p. 316)
API inventory (p. 50)
Argument tables (p. 320)

B

Background property (p. 11)
Bindings, deferred (p. 330)
Bindings, native (p. 22)
Button actions (p. 160)
Button grids (p. 162)
ButtonGroup (p. 156)
ButtonGroup:GetHtml() (p. 166)
ButtonGroup:Id() (p. 167)
ButtonGroup:PrepareContent() (p. 165)
ButtonGroup:WithContent() (p. 158)
ButtonGroup.New() (p. 157)

C

Callbacks (p. 320)
Checkbox values (p. 141)
CheckboxGroup (p. 137)
CheckboxGroup:GetHtml() (p. 149)
CheckboxGroup:GetValues() (p. 141)
CheckboxGroup:Id() (p. 150)
CheckboxGroup:PrepareContent() (p. 149)
CheckboxGroup:SetValues() (p. 143)
CheckboxGroup:WithContent() (p. 139)
CheckboxGroup.New() (p. 138)
Cleanup (p. 28)
Clipboard (p. 68)
Combobox (p. 192)
Combobox:AddItem() (p. 201)
Combobox:AllowEmptySelection() (p. 197)
Combobox:Clear() (p. 203)
Combobox:Count() (p. 204)
Combobox:GetHtml() (p. 207)
Combobox:GetItems() (p. 203)
Combobox:Id() (p. 208)
Combobox:PrepareContent() (p. 206)
Combobox:RemoveItem() (p. 202)
Combobox:ReplaceItem() (p. 202)
Combobox:Select() (p. 196)
Combobox:Selected() (p. 198)
Combobox:SelectedValue() (p. 199)
Combobox:SetItems() (p. 200)
Combobox:WithContent() (p. 194)
Combobox.New() (p. 193)
Compiled modules (.gfc) (p. 38)
Component construction (p. 115)
Component libraries (p. 37)
Components (p. 113)
Content preparation (p. 117)
Control identifiers (p. 63)
Core services (p. 53)
Core.AddCSS() (p. 53)
Core.AddHandler() (p. 53)
Core.ComponentId() (p. 53)
Core.FileExists() (p. 53)

Core.InstanceCounter() (p. 53)
Core.NumberFormatter() (p. 53)
Core.RegisterContent() (p. 53)
CSS considerations (p. 67)
Custom completion buttons (p. 295)

D

DataPicker (p. 246)
DataPicker:Enabled() (p. 248)
DataPicker:GetHtml() (p. 251)
DataPicker:GetText() (p. 247)
DataPicker:Id() (p. 250)
DataPicker:IsValid() (p. 248)
DataPicker:PrepareContent() (p. 251)
DataPicker:Refresh() (p. 250)
DataPicker:WithContent() (p. 247)
DataPicker.New() (p. 246)
Default buttons (p. 288)
Details.ini (p. 267)
Dialog completion (p. 288)
DiscardResults() (p. 26)
Document readiness (p. 67)
DOM updates (p. 364)
Dot and colon notation (p. 115)
Duplicate library loading (p. 43)
Dynamic HTML (p. 364)

E

Empty selection (p. 197)
Error handling (p. 327)
EscapeHtml() (p. 400)
Event dispatch (p. 323)
Examples (p. 380)
Examples - A Dimension Calculator (p. 385)
Examples - A First Dialog (p. 382)
Examples - A Preview Workbench (p. 396)
Examples - Choices and Options (p. 389)
Examples - Dependent Lists (p. 392)
Examples - The Layout Grid (p. 401)
ExecutionEnabled() (p. 289)

F

Field changes (p. 96)
Field classes (p. 96)
Field primitives (p. 93)
File filtering (p. 260)
File selection (p. 263)
FileList (p. 257)
FileList:GetHtml() (p. 271)
FileList:Id() (p. 268)
FileList:PrepareContent() (p. 271)
FileList:Selected() (p. 262)
FileList:WithContent() (p. 259)
FileList.New() (p. 258)
FinalizeScript() (p. 27)
Fluent construction (p. 115)
Forge (p. 4)
Framework contracts (p. 62)
Framework overview (p. 7)

G

Gadget Factory (p. 2)
Gadget lifecycle (p. 15)
GadgetDef (p. 10)
GetDivHtml() (p. 356)
GetDoubleFieldHtml() (p. 101)
GetElementsHtml() (p. 351)
GetIntegerFieldHtml() (p. 101)
GetLabelFieldHtml() (p. 101)
GetLabelHtml() (p. 90)
GetLabelSpanHtml() (p. 106)
GetStateFlagHtml() (p. 348)
GetTextFieldHtml() (p. 101)
GF.Dialog (p. 23)
GF.GadgetDef (p. 10)

H

HTML generation (p. 351)
HTML primitives (p. 71)
HTML_Dialog (p. 21)

I

InitializeDialog(dlg) (p. 49)
InitializeScript(job, script_path) (p. 34)
Installation (p. 4)
Item identity (p. 239)

J

JavaScript insertion (p. 375)
Job requirement (p. 11)

K

Keyboard navigation (p. 230)

L

Label updates (p. 104)
LayerPicker (p. 252)
LayerPicker:ActivateLayer() (p. 256)
LayerPicker:Enabled() (p. 253)
LayerPicker:GetText() (p. 253)
LayerPicker:Id() (p. 255)
LayerPicker:IsValid() (p. 254)
LayerPicker:RefreshLayers() (p. 255)
LayerPicker:WithContent() (p. 253)
LayerPicker.New() (p. 252)
Layout containers (p. 83)
Layout Grid (p. 401)
Library dependencies (p. 45)
ListBox (p. 217)
ListBox:AddItem() (p. 227)
ListBox:Clear() (p. 228)
ListBox:Count() (p. 229)
ListBox:GetHtml() (p. 234)
ListBox:GetItems() (p. 228)
ListBox:Id() (p. 235)
ListBox:PrepareContent() (p. 233)
ListBox:RemoveItem() (p. 227)
ListBox:ReplaceItem() (p. 228)
ListBox:Select() (p. 222)
ListBox:Selected() (p. 223)
ListBox:SetItems() (p. 225)
ListBox:WithContent() (p. 219)
ListBox.New() (p. 218)
LoadComponents() (p. 36)
Loading Gadget Factory (p. 9)
LuaButton callbacks (p. 323)
LuaButton class requirements (p. 371)

M

main(script_path) (p. 17)

N

NoDefaultButtons() (p. 293)
Number formatting (p. 281)
NumberFormatter() (p. 281)
Numeric precision (p. 286)

O

OnError(err) (p. 327)
OnLuaButton_XXXX(id) (p. 56)
OnValidityChanged(valid) - DataPicker (p. 249)
OnValidityChanged(valid) - LayerPicker (p. 254)
Optional arguments (p. 94)

P

Post-load initialization (p. 367)
ProcessResults() (p. 25)
Programming conventions (p. 59)

Q

Quick start (p. 4)

R

RadioGroup (p. 120)
RadioGroup:GetHtml() (p. 129)
RadioGroup:Id() (p. 130)
RadioGroup:PrepareContent() (p. 129)
RadioGroup:Select() (p. 123)
RadioGroup:Selected() (p. 125)
RadioGroup:WithContent() (p. 122)
RadioGroup.New() (p. 121)
Read-only rendering (p. 353)

Registration (p. 118)
ReplaceHandler() (p. 322)

S

script_path (p. 17)
Selection callbacks (p. 116)
SetInnerHtml() - HTML_Dialog (p. 364)
SetOuterHtml() - HTML_Dialog (p. 365)
ShowDialog() - HTML_Dialog (p. 24)
State flags (p. 340)
StatusBar (p. 298)
StatusBar.Caution() (p. 304)
StatusBar.GetHtml() (p. 308)
StatusBar.Init() (p. 298)
StatusBar.Normal() (p. 304)
StatusBar.PrepareContent() (p. 308)
StatusBar.Update() (p. 302)
StatusBar.Warning() (p. 304)
Stylesheet order (p. 43)

T

ToolStrip (p. 173)
ToolStrip docking (p. 180)
ToolStrip:GetHtml() (p. 183)
ToolStrip:Id() (p. 185)
ToolStrip:PrepareContent() (p. 183)
ToolStrip:WithContent() (p. 176)
ToolStrip.New() (p. 174)
Trident limitations (p. 66)

U

UpdateLabelField() - HTML_Dialog (p. 104)
UpdateTextField() - picker refresh requirement (p. 250)
Utilities (p. 278)

V

Value types (p. 116)
Vectric entry points (p. 56)

Z

Zero decimal places (p. 286)