RFC-031: Module registry and construction context for wtf

- **Status:** draft, open for review
- **Author:** J. Appleseed
- **Target:** `wtfutil/wtf` at `bb838c1`
- **Affects:** `app/widget_maker.go`, `wtf/`, every package under `modules/`

1. Motivation

Adding a module requires editing both the imports and the `MakeWidget` switch in `app/widget_maker.go:9–379`. Contributors adding unrelated modules can conflict in this file. The switch also has to know each module's constructor arguments, including whether it needs `pages` or build metadata.

This RFC replaces the switch with a registry: each module registers a function that creates its widget under a type name. `MakeWidget` looks up that function and calls it. A shared `ModuleContext` gives constructors the same set of application values.

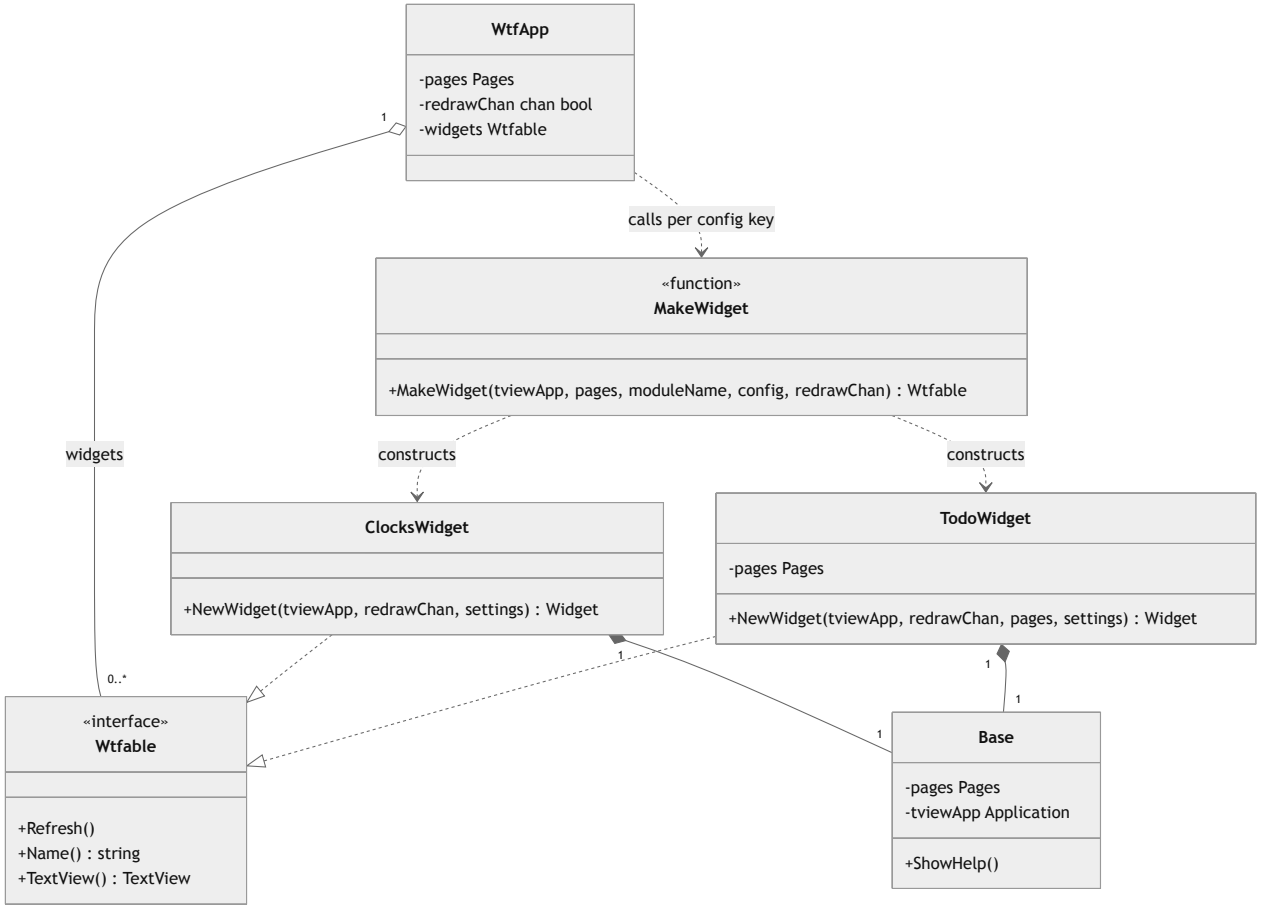
New modules would need a registration function and an import in `modules/all.go`, without changes to the `app` package. Modules in other repositories could use the same registration mechanism.

2. Current design

`MakeWidgets` calls `MakeWidget` for each entry in `wtf.mods`. `MakeWidget` returns `nil` for missing or disabled entries. Otherwise, it selects the module using `moduleConfig.UString("type", moduleName)`, builds its settings, and calls its constructor (`app/widget_maker.go:98–399`).

Constructors take either the application and redraw channel, or those values plus `pages`. The `system` module also takes build metadata. Modules can pass `nil` for pages to the shared view code; six modules also store pages for their own dialogs.

An unrecognized type produces an `unknown` widget that displays an error (`modules/unknown/widget.go:29`).



3. Proposal

3.1 Registry API

Add a map from module type names to factory functions (functions that create widgets) in the `wtf` package, next to `wtf/wtfable.go`.

```
package wtf

// ModuleFactory builds one widget from its configuration.
type ModuleFactory func(
        ctx *ModuleContext,
        moduleName string,
        moduleConfig *config.Config,
        globalConfig *config.Config,
) Wtfable

var registry = map[string]ModuleFactory{}

// RegisterModule records a factory under a module type name.
func RegisterModule(name string, factory ModuleFactory) {
        registry[name] = factory
}

// LookupModule returns the factory registered for a module type name.
func LookupModule(name string) (ModuleFactory, bool) {
        factory, ok := registry[name]
        return factory, ok
}
```

Only `RegisterModule` writes to the map. Keeping the registry in `wtf` lets modules register without importing `app`, which would create an import cycle.

3.2 Factory signature and ModuleContext

Group the application, pages, and redraw channel in one struct:

```
package wtf

// ModuleContext carries the application-owned collaborators that a module
// may need while it is being constructed.
type ModuleContext struct {
        TViewApp   *tview.Application
        Pages      *tview.Pages
        RedrawChan chan bool
}
```

Each module's `NewWidget` takes this context and its settings, using the fields it needs. For `clocks`:

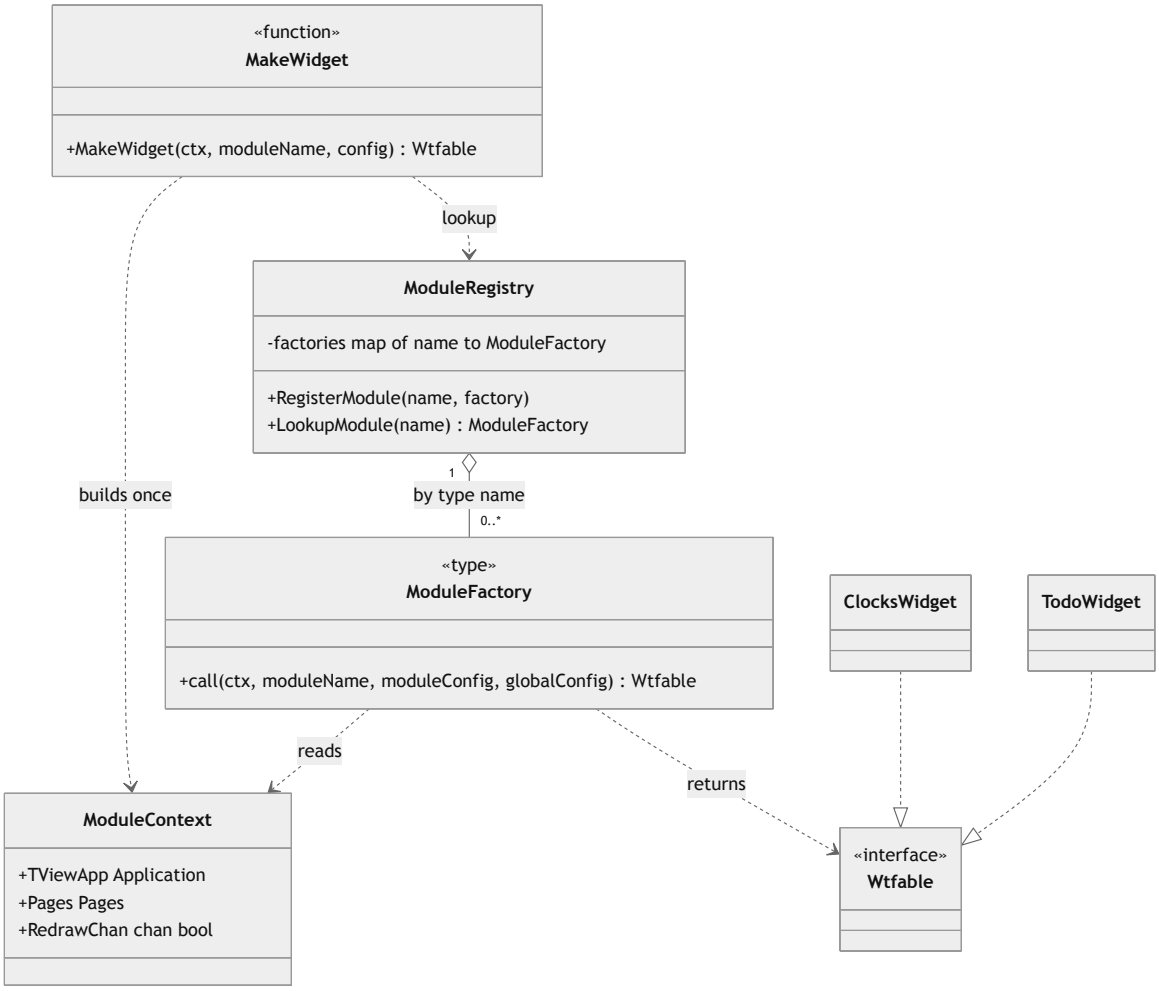
```
package clocks

func NewWidget(ctx *wtf.ModuleContext, settings *Settings) *Widget {
        widget := Widget{
                TextWidget: view.NewTextWidget(
                        ctx.TViewApp, ctx.RedrawChan, ctx.Pages, settings.Common,
                ),

                settings:   settings,
                dateFormat: settings.dateFormat,
                timeFormat: settings.timeFormat,
        }

        widget.clockColl = widget.buildClockCollection()

        return &widget
}
```



3.3 Registration and imports

Each module registers its factory in `register.go`. Go runs the imported package's `init()` function once before `main`:

```
package clocks

func init() {
        wtf.RegisterModule("clocks", func(
                ctx *wtf.ModuleContext,
                moduleName string,
                moduleConfig *config.Config,
                globalConfig *config.Config,
        ) wtf.Wtfable {
                settings := NewSettingsFromYAML(moduleName, moduleConfig,
globalConfig)
                return NewWidget(ctx, settings)
        })
}
```

A new `modules/all.go` imports every built-in module. The _ imports run each package's initialization without using its exported names:

```
// Package modules imports every built-in module for its registration
// side effect.
package modules

import (
        _ "github.com/wtfutil/wtf/modules/airbrake"
        _ "github.com/wtfutil/wtf/modules/asana"
        _ "github.com/wtfutil/wtf/modules/azuredevops"
        // ... one line per built-in module ...
        _ "github.com/wtfutil/wtf/modules/zendesk"
)
```

`main.go` blank-imports `modules`. The `app` package no longer needs module imports.

3.4 Unknown types

`MakeWidget` returns an error if no factory matches the configured type:

```
func MakeWidget(
        ctx *wtf.ModuleContext,
        moduleName string,
        config *config.Config,
) (wtf.Wtfable, error) {
        moduleConfig, _ := config.Get("wtf.mods." + moduleName)
        if moduleConfig == nil {
                return nil, nil
        }

        if enabled := moduleConfig.UBool("enabled", false); !enabled {
                return nil, nil
        }

        typeName := moduleConfig.UString("type", moduleName)

        factory, ok := wtf.LookupModule(typeName)
        if !ok {
                return nil, fmt.Errorf(
                        "module %q: no module registered for type %q", moduleName,
typeName,
                )
        }

        return factory(ctx, moduleName, moduleConfig, config), nil
}
```

`MakeWidgets` collects these errors. `NewWtfApp` prints them and exits, following `ModuleValidator.Validate` (`app/module_validator.go:27–41`). Unknown types become startup errors, and the `unknown` package is deleted.

3.5 Tests

Update `app/widget_maker_test.go` to pass a `*wtf.ModuleContext` and check the returned error. Keep the existing widget-type assertions. Tests can also register fake factories to check `MakeWidget` without constructing real modules.

4. Migration plan

Add the registry, `ModuleContext`, and revised `MakeWidget`, keeping the switch as a fallback until all modules have been converted. This lets the application keep working during migration.

Convert roughly ten modules per PR: update `NewWidget`, add `register.go` and a blank import, and remove the module's switch case.

For `system`, move `buildDate()` and `buildVersion()` (`app/widget_maker.go:403–437`) to an exported package that the module can call directly.

After converting all modules, remove the switch, its imports, and the `unknown` package. Update `generator/textwidget.go:40` to tell authors to add an import to `modules/all.go`.

5. Alternatives considered

- Keep the switch: it is easy to read, but `app` still needs to import and construct every module.
- Generate the switch with `go:generate`: this removes manual edits but adds a generation step and a file that contributors must keep up to date.
- Use a central map: this simplifies the switch but keeps the shared list and module imports in `app`.
- Load modules with Go's `plugin` package: this requires separate builds and matching toolchain and dependency versions. It also lacks Windows support.

6. Open questions

- Should the registration function be called `RegisterModule` or `Register`?
- Should the import list live in `modules/all.go` or `app/modules.go`?
- Should `ModuleContext` be a struct or an interface? The struct is simple, but would an interface help tests?
- Should `MakeWidgets` return `([]wtf.Wtfable, []error)` so callers can handle partial failures?
- Should a lint check keep the import list alphabetized?
- Should each migration PR convert five modules or ten?