Render Your Go Code Clean:

Introduction to Dependency Injection with Fx

RenderATL 2024
Workshop

Uber

Agenda

01 Introduction (5 minutes)

02 Dependency Injection (20 minutes)

03 Fx Framework (25 minutes)

- - - Break: 10 minutes - - -

04 Project Overview (5 minutes)

05 Hands on time!

06 Wrap up + feedback (5 minutes)

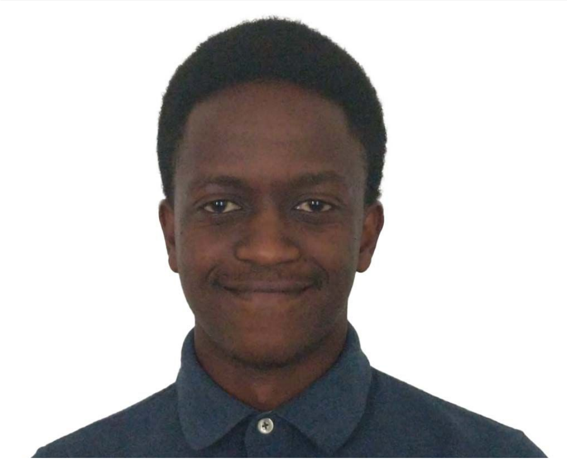
Meet your team!

Dorian Perkins

Staff Software Engineer
Software Networking team

Kemet Dugue

Software Engineer
Driver Onboarding team



Paul Murage

Software Engineer
Configuration Platform team

What to expect in this workshop

What you’ll need ...

Laptop

You'll be coding in this workshop, so make sure you have your trusty laptop handy.

Go

Ensure you have Go downloaded and installed locally before the workshop.

[https://go.dev/doc/install]

IDE

Get your favorite code editor ready to go and ensure it is set up to work with Go.

(prior experience in Go is not required).

What you'll get ...

Understanding of dependency injection

Introduction to the Fx application framework

Hands-on experience writing a Go application with Fx

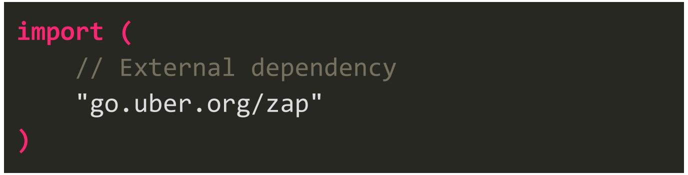
Dependency Injection

What is a dependency?

Code that is relied on by other code to function correctly

External dependencies

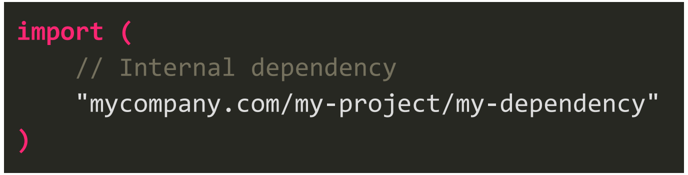
- Pre-written code created by a third-party (i.e., libraries or frameworks)



```go
import (
    // External dependency
    "go.uber.org/zap"
)
```

Internal dependencies

- Connections between different parts of your own code

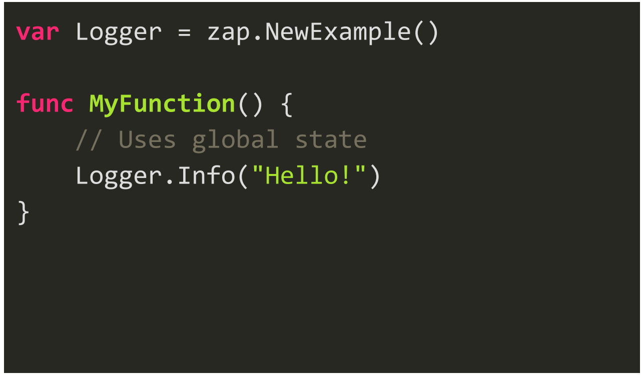


```go
import (
    // Internal dependency
    "mycompany.com/my-project/my-dependency"
)
```

What is dependency injection (DI)?

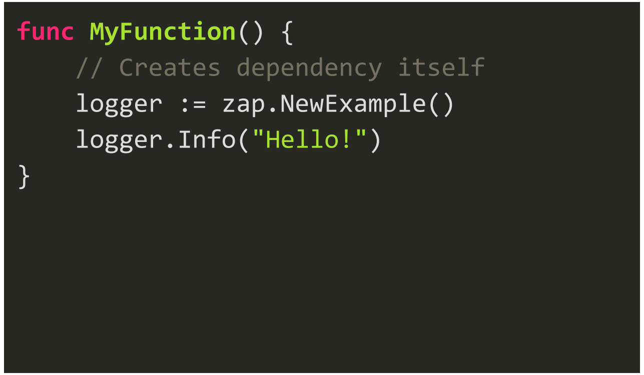
Supplying an object with its dependencies rather than creating them itself

❌ Example 1 (global state)



```go
var Logger = zap.NewExample()

func MyFunction() {
    // Uses global state
    Logger.Info("Hello!")
}
```

❌ Example 2 (create deps)

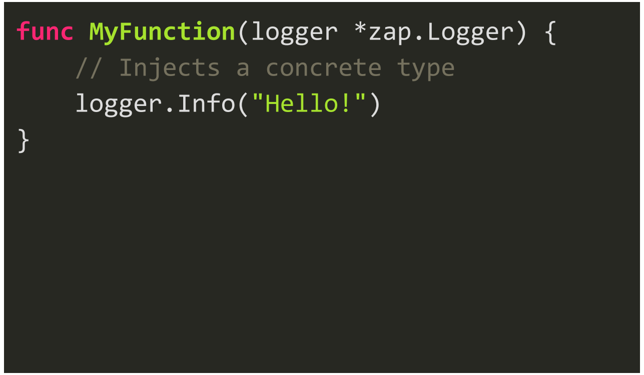


```go
func MyFunction() {
    // Creates dependency itself
    logger := zap.NewExample()
    logger.Info("Hello!")
}
```

What is dependency injection (DI)?

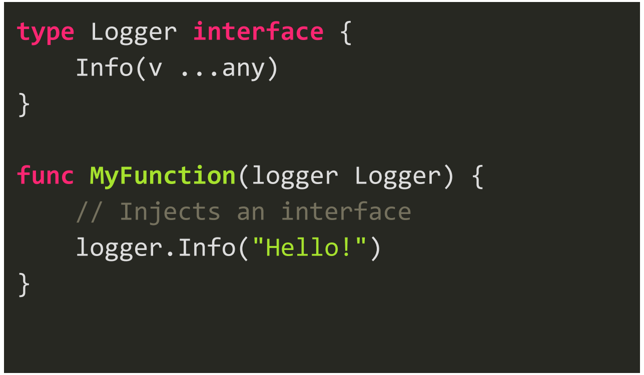
Supplying an object with its dependencies rather than creating them itself

✔ Example 3 (DI, concrete type)



```
func MyFunction(logger *zap.Logger) {
    // Injects a concrete type
    logger.Info("Hello!")
}
```

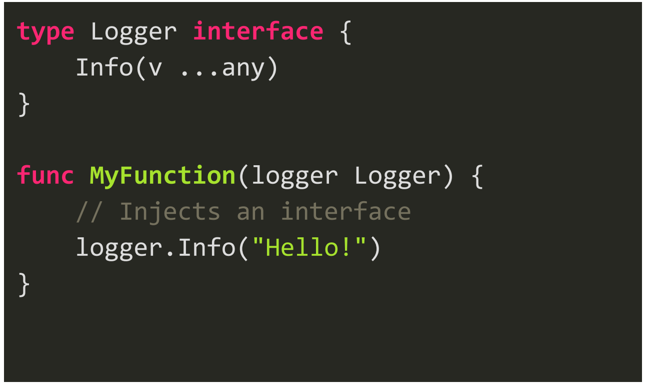
✔ Example 4 (DI, interface)



```
type Logger interface {
    Info(v ...any)
}

func MyFunction(logger Logger) {
    // Injects an interface
    logger.Info("Hello!")
}
```

Benefits

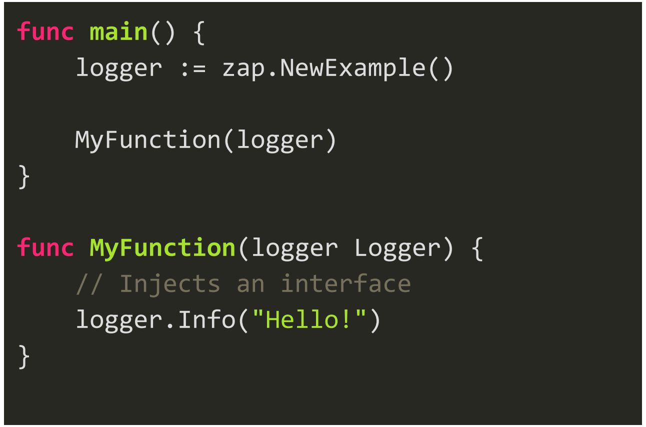
- **Loose coupling**
  - Objects are less reliant on specific implementations



```go
type Logger interface {
    Info(v ...any)
}

func MyFunction(logger Logger) {
    // Injects an interface
    logger.Info("Hello!")
}
```

Benefits

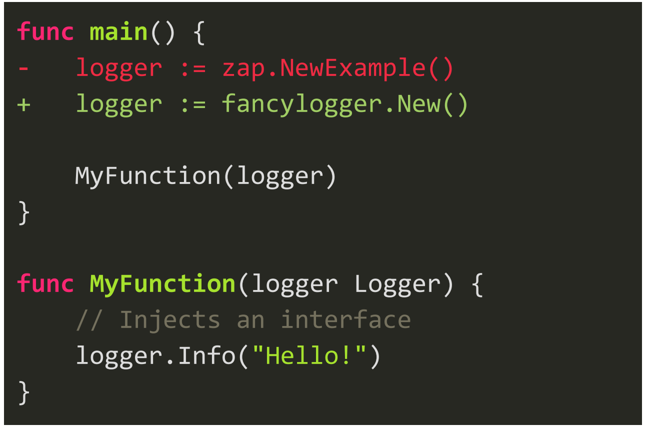
- **Loose coupling**
- **Promotes modularity**
  - Separates concerns of dependency creation and usage



```go
func main() {
    logger := zap.NewExample()

    MyFunction(logger)
}

func MyFunction(logger Logger) {
    // Injects an interface
    logger.Info("Hello!")
}
```

Benefits

- **Loose coupling**
- **Promotes modularity**
- **Increased maintainability**
  - Easier to swap out implementations without code changes



```
func main() {
-   logger := zap.NewExample()
+   logger := fancylogger.New()

    MyFunction(logger)
}

func MyFunction(logger Logger) {
    // Injects an interface
    logger.Info("Hello!")
}
```

Benefits

- Loose coupling
- Promotes modularity
- Increased maintainability
- Improved testability
    - Dependencies can be easily mocked or stubbed

```go
type Logger interface {
    Info(v ...any)
}
```

```go
func Test_MyFunction(t *testing.T) {
    logger := zaptest.NewLogger(t)
    MyFunction(logger)
    ...
}

func MyFunction(logger Logger) {
    // Injects an interface
    logger.Info("Hello!")
}
```

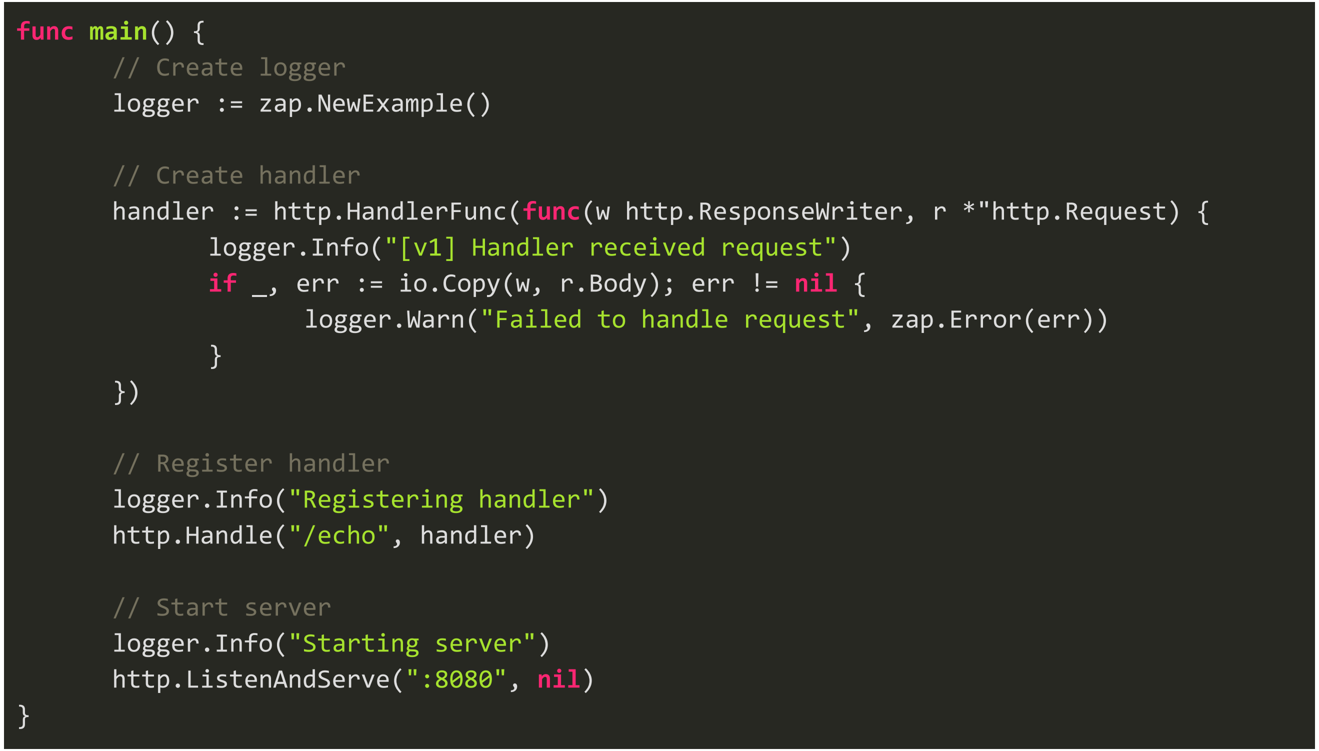
```go
func Test_MyFunction(t *testing.T) {
    logger := mocks.NewMockLogger()
    MyFunction(logger)
    ...
}

func MyFunction(logger Logger) {
    // Injects an interface
    logger.Info("Hello!")
}
```

DI: Examples & Live Demo

(code examples)

https://t.uber.com/render-demos

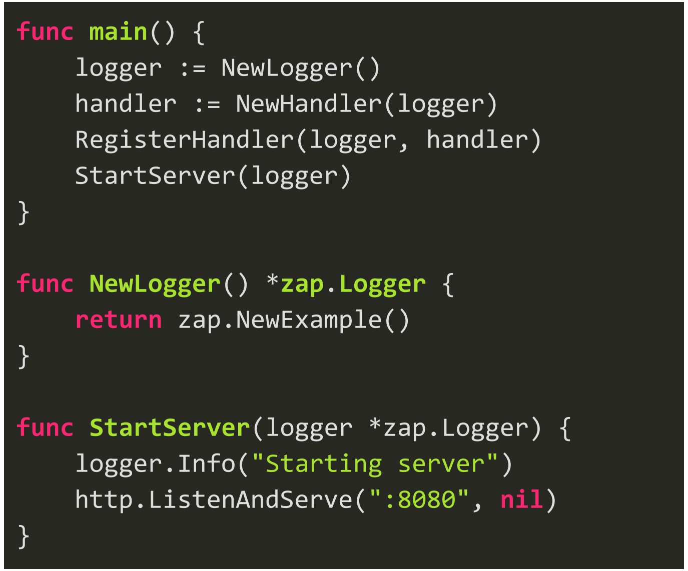
v1 – No dependency injection



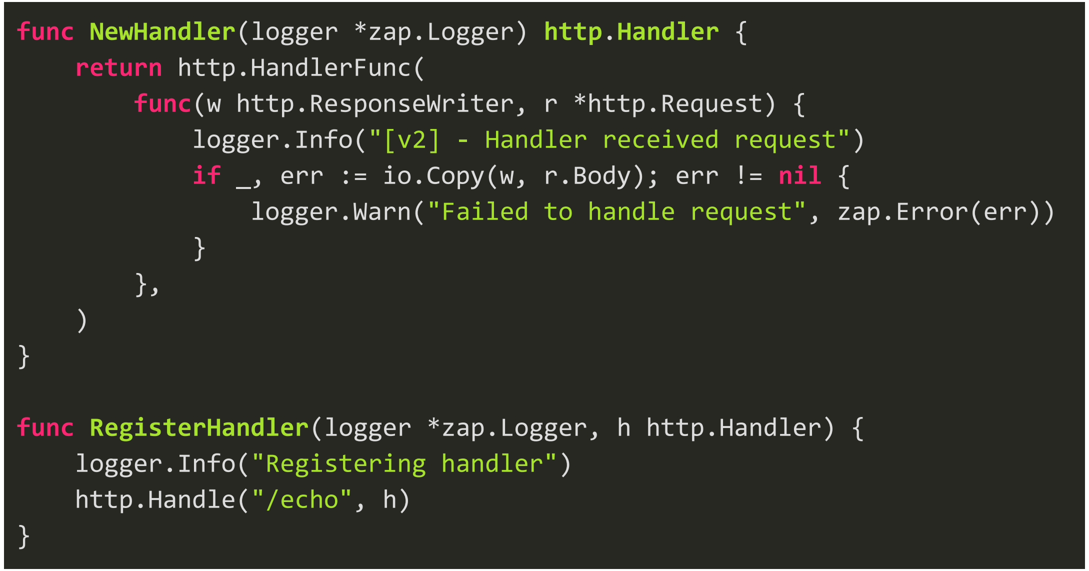
```
func main() {
    // Create logger
    logger := zap.NewExample()

    // Create handler
    handler := http.HandlerFunc(func(w http.ResponseWriter, r *"http.Request) {
        logger.Info("[v1] Handler received request")
        if _, err := io.Copy(w, r.Body); err != nil {
            logger.Warn("Failed to handle request", zap.Error(err))
        }
    })

    // Register handler
    logger.Info("Registering handler")
    http.Handle("/echo", handler)

    // Start server
    logger.Info("Starting server")
    http.ListenAndServe(":8080", nil)
}
```

Demo (v1)

v2 – Manual dependency injection



```go
func main() {
    logger := NewLogger()
    handler := NewHandler(logger)
    RegisterHandler(logger, handler)
    StartServer(logger)
}

func NewLogger() *zap.Logger {
    return zap.NewExample()
}

func StartServer(logger *zap.Logger) {
    logger.Info("Starting server")
    http.ListenAndServe(":8080", nil)
}
```



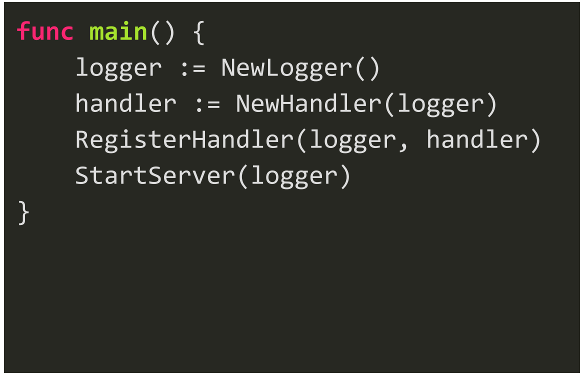
```go
func NewHandler(logger *zap.Logger) http.Handler {
    return http.HandlerFunc(
        func(w http.ResponseWriter, r *http.Request) {
            logger.Info("[v2] - Handler received request")
            if _, err := io.Copy(w, r.Body); err != nil {
                logger.Warn("Failed to handle request", zap.Error(err))
            }
        },
    )
}

func RegisterHandler(logger *zap.Logger, h http.Handler) {
    logger.Info("Registering handler")
    http.Handle("/echo", h)
}
```

Demo (v2)

Cost of manual dependency injection

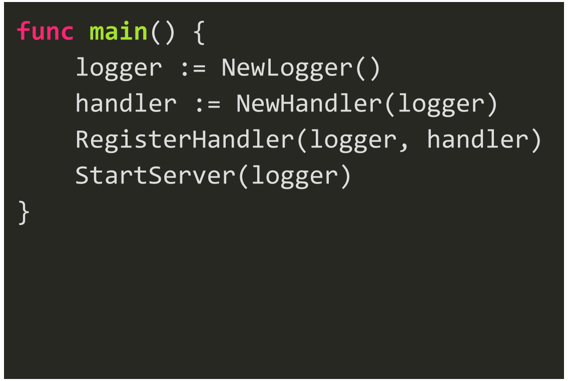
- **Requires writing boilerplate in every service**
  - Repetitive and time consuming

App A



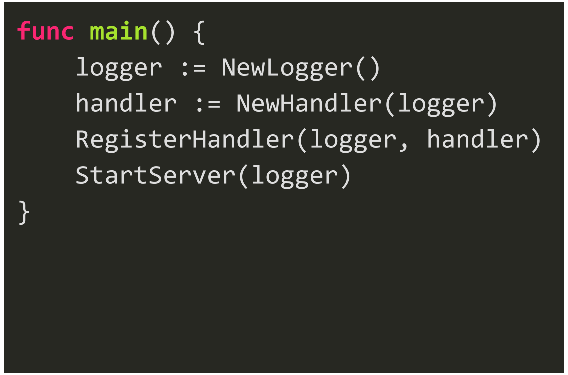
```go
func main() {
    logger := NewLogger()
    handler := NewHandler(logger)
    RegisterHandler(logger, handler)
    StartServer(logger)
}
```

App B



```go
func main() {
    logger := NewLogger()
    handler := NewHandler(logger)
    RegisterHandler(logger, handler)
    StartServer(logger)
}
```

App C

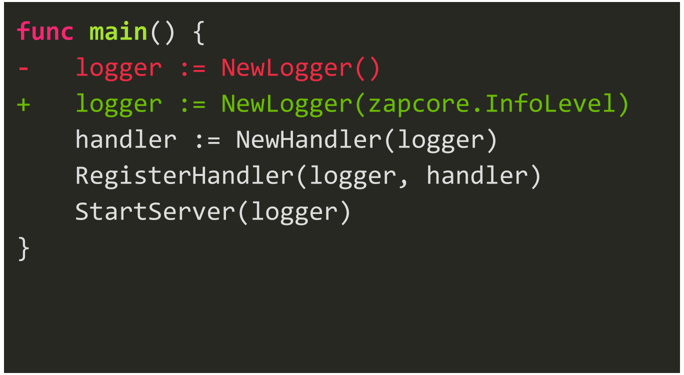


```go
func main() {
    logger := NewLogger()
    handler := NewHandler(logger)
    RegisterHandler(logger, handler)
    StartServer(logger)
}
```

Cost of manual dependency injection

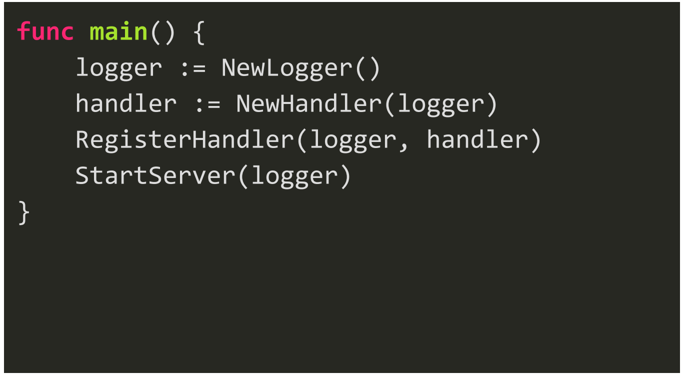
- **Requires writing boilerplate in every service**
  - Repetitive and time consuming
- **Long-term maintenance burden as application evolves**
  - Some adopt quickly, others fall behind; usages diverge over time

App A (adopts change)



```
func main() {
-   logger := NewLogger()
+   logger := NewLogger(zapcore.InfoLevel)
    handler := NewHandler(logger)
    RegisterHandler(logger, handler)
    StartServer(logger)
}
```

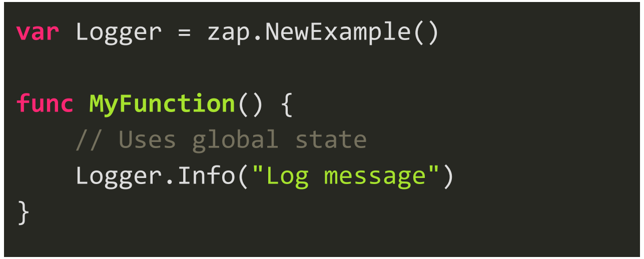
App B (falls behind)



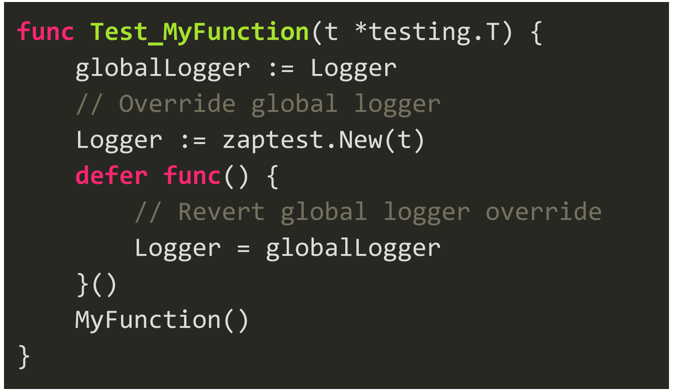
```
func main() {
    logger := NewLogger()
    handler := NewHandler(logger)
    RegisterHandler(logger, handler)
    StartServer(logger)
}
```

Cost of manual dependency injection

- **Requires writing boilerplate in every service**
  - Repetitive and time consuming
- **Long-term maintenance burden as application evolves**
  - Some adopt quickly, others fall behind; usages diverge over time
- **Can lead to creation of global state**
  - Less effort to maintain (singleton); complicates testing



```go
var Logger = zap.NewExample()

func MyFunction() {
    // Uses global state
    Logger.Info("Log message")
}
```



```go
func Test_MyFunction(t *testing.T) {
    globalLogger := Logger
    // Override global logger
    Logger := zaptest.New(t)
    defer func() {
        // Revert global logger override
        Logger = globalLogger
    }()
    MyFunction()
}
```

Cost of manual dependency injection

- **Requires writing boilerplate in every service**
  - Repetitive and time consuming
- **Long-term maintenance burden as application evolves**
  - Some adopt quickly, others fall behind; usages diverge over time
- **Can lead to creation of global state**
  - Less effort to maintain (singleton); complicates testing
- **Cost multiplies at scale**
  - Uber’s hypergrowth demanded smarter, more-efficient solution

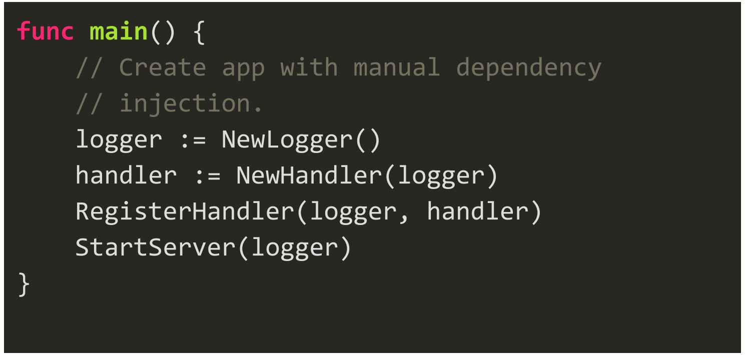
Fx

github.com/uber-go/fx

What is Fx?

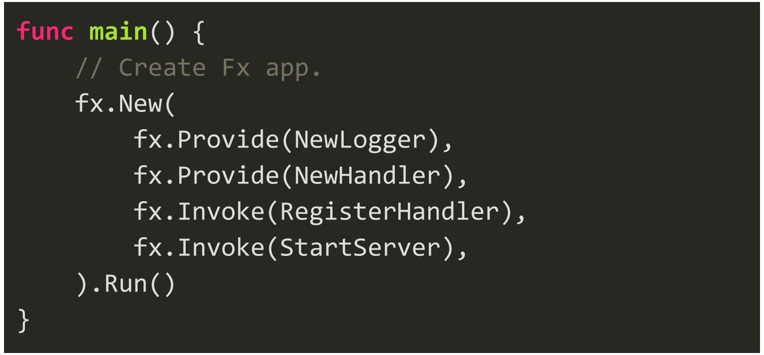
- A dependency injection framework for Go, built and battle-tested at **Uber**.
- Provides dependency injection without the manual wiring.

v2 – Manual DI



```go
func main() {
    // Create app with manual dependency
    // injection.
    logger := NewLogger()
    handler := NewHandler(logger)
    RegisterHandler(logger, handler)
    StartServer(logger)
}
```

v3 – Fx



```go
func main() {
    // Create Fx app.
    fx.New(
        fx.Provide(NewLogger),
        fx.Provide(NewHandler),
        fx.Invoke(RegisterHandler),
        fx.Invoke(StartServer),
    ).Run()
}
```

Demo(v3)

The magic

Connecting providers to receivers

Providers

“Here’s an instance of component X”

```go
// NewLogger returns a logger.
func NewLogger() *log.Logger {
    return log.New(os.Stdout, "", 0)
}
```

Receivers

“I need an instance of component X”

```go
// NewHandler receives a logger as a dependency.
func NewHandler(logger *log.Logger) http.Handler {
    logger.Print("Log message")
}
```

Provide & Invoke

Core building blocks

Provide

“Registers a function with Fx lifecycle”

```
func main() {
    fx.New(
        fx.Provide(NewLogger)
        ...
    ).Run()
}
```

Invoke

“Executes a function during Fx lifecycle”

```
func main() {
    fx.New(
        fx.Invoke(NewLogger)
        ...
    ).Run()
}
```

- **`Provide`**s are only executed <u>as necessary</u> (i.e., if they have a receiver).
- **`Invoke`**s are <u>always</u> executed.

Provide & Invoke

Fx’s core building blocks

Revisiting example v3

Why Provide vs Invoke?

```
func main() {
    // Create Fx app.
    fx.New(
        fx.Provide(NewLogger),
        fx.Provide(NewHandler),
        fx.Invoke(RegisterHandler),
        fx.Invoke(StartServer),
    ).Run()
}
```

No return values → No receivers

```
func RegisterHandler(h http.Handler) {
    http.Handle("/echo", h)
}

func StartServer() {
    http.ListenAndServe(":8080", nil)
}
```

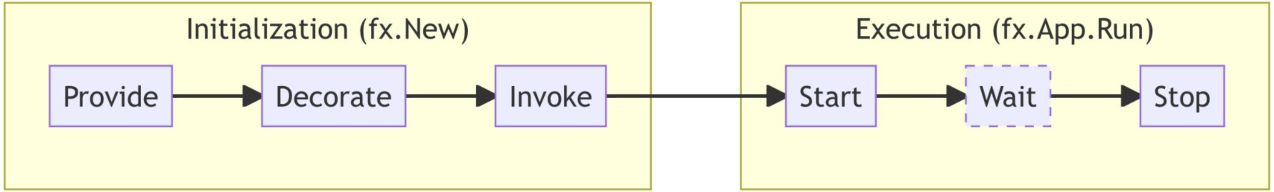
- Core business logic is typically **`invoke`**d.

Fx Lifecycle

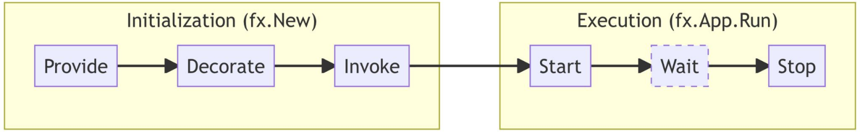
Two high-level phases: *initialization* and *execution*.

Initialization: register constructors and decorators, run invoked functions

Execution: run all startup hooks, wait for stop signal , run all shutdown hooks



Lifecycle Hooks



Lifecycle hooks provide the ability to schedule work to be executed by Fx when the application starts up or shuts down.

Fx allows **two** kinds of hooks:

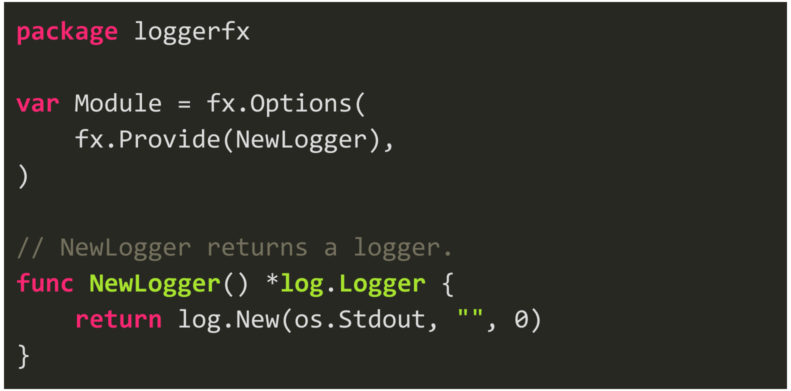
- **`OnStart`** hooks, run in the order they were appended at *Start*
  - *Example: Start HTTP server*
- **`OnStop`** hooks, run in the reverse order they were appended at *Stop*
  - *Example: Shutdown HTTP server*

Modules

Sharable bundles of one or more components

Logger as a Module

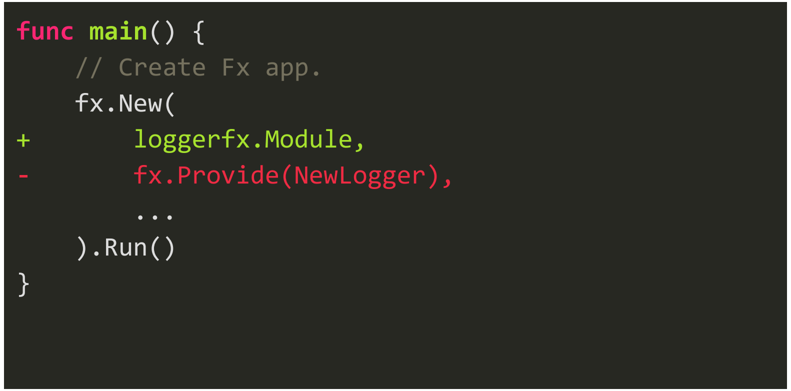
Library module names should end in -fx



```go
package loggerfx

var Module = fx.Options(
    fx.Provide(NewLogger),
)

// NewLogger returns a logger.
func NewLogger() *log.Logger {
    return log.New(os.Stdout, "", 0)
}
```

Using Logger Module

Replace fx.Provide with Fx module

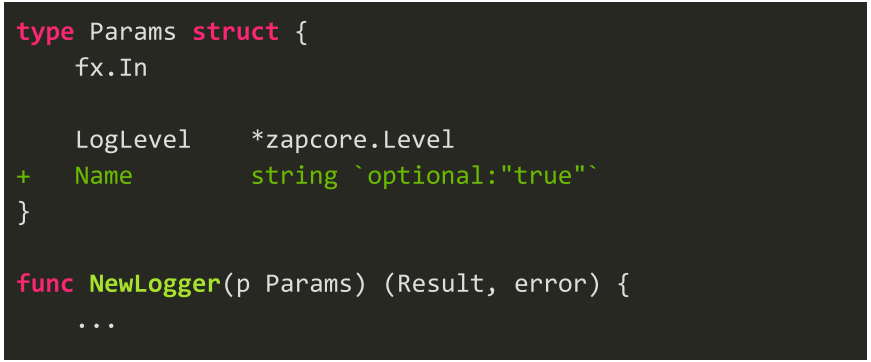


```
func main() {
    // Create Fx app.
    fx.New(
+       loggerfx.Module,
-       fx.Provide(NewLogger),
        ...
    ).Run()
}
```

Parameter objects

Functions exposed by a module should not accept dependencies directly as parameters. Instead, they should use a **parameter object.**

This allows new *optional* dependencies to be added in a backwards-compatible manner.

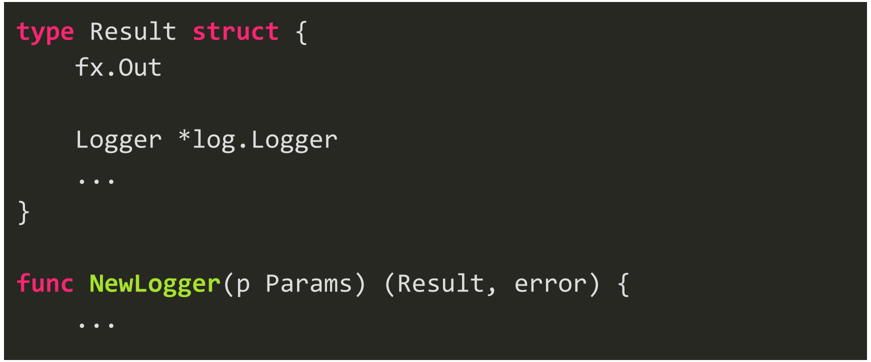


```
type Params struct {
    fx.In

    LogLevel    *zapcore.Level
+   Name        string `optional:"true"`
}

func NewLogger(p Params) (Result, error) {
    ...
```

Result objects

Functions exposed by a module should not declare their results as regular return values. Instead, they should use a **result object.**

This allows new results to be added in a backwards-compatible manner.



```go
type Result struct {
    fx.Out

    Logger *log.Logger
    ...
}

func NewLogger(p Params) (Result, error) {
    ...
```

Modules

Sharable bundles of one or more components

Module bundle

Provide...all the things!

```
package uberfx

var Module = fx.Options(
    loggerfx.Module,
    metricsfx.Module,
    rpcfx.Module,
    serverfx.Module,
    storagefx.Module,
    ...
)
```

Using Module bundle*

Complex scaffolding made easy

```
func main() {
    // Create Fx app.
    fx.New(
        uberfx.Module,
        ...
    ).Run()
}
```

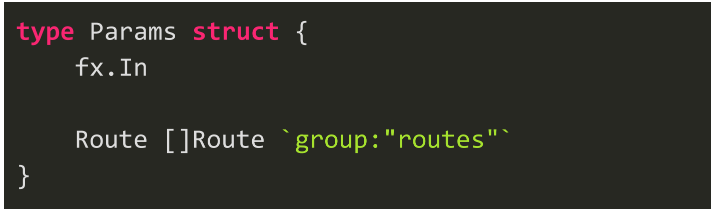
* Useful for adding/deprecating shared libraries without modifying **`main`**.

Value Groups

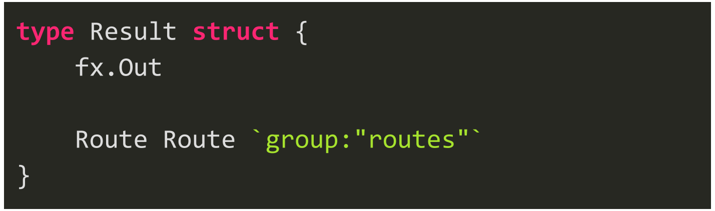
Fx does not allow two instances of the same type to be present in the container.

A value group is...

- a *collection* of values of the same type.
- defined using the “group” *annotation*.
    - Must be used on both the input parameter slice and output result.



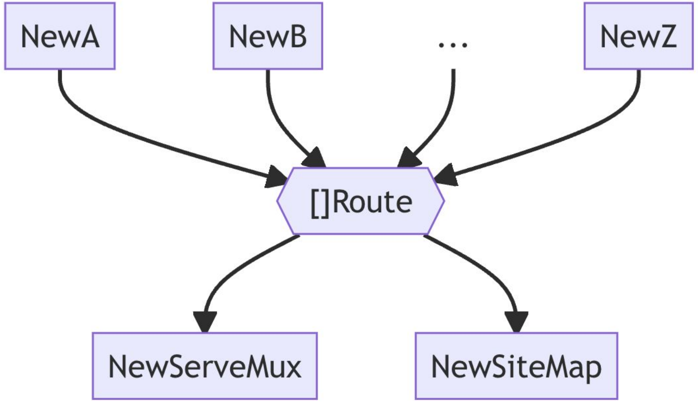
```go
type Params struct {
    fx.In

    Route []Route `group:"routes"`
}
```



```go
type Result struct {
    fx.Out

    Route Route `group:"routes"`
}
```

Value Groups

- Any number of constructors can feed values into a value group.
- Any number of consumers can read from a value group.



Pros

- **Eliminate globals**
  - Helps remove globally shared state
- **Increase efficiency**
  - Less boilerplate code → Less repetitive work
- **No manual wiring**
  - Eliminates need to manually wire up dependencies
- **Code reuse**
  - Build loosely coupled, well-integrated sharable modules

Cons

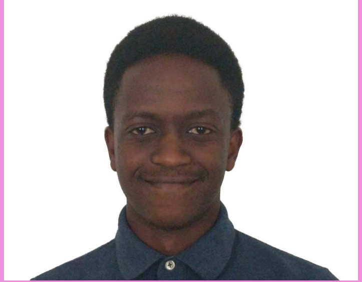
- **Steeper learning curve**
  - Introduces complexity harder to grasp for new developers
- **Loss of control flow**
  - Framework controls order of execution
- **Harder to debug**
  - Missing dependencies become runtime errors

Q&A

Thank you!

Dorian Perkins

Kemet Dugue



Paul Murage