Golang FX

https://pkg.go.dev/go.uber.org/fx
https://uber-go.github.io/fx/

O que é

É uma biblioteca que facilita a construção de aplicações que dependem de injeções de dependências e gerenciamento de componentes, oferecendo de maneira estruturada e modular a possibilidade de organização de código.

Principais Funcionalidades

Injeção de Dependências

Declaração de dependências de forma declarativa, evitando a criação manual de instâncias de componentes e a gerenciar suas dependências de maneira mais eficiente.

Principais Funcionalidades

Ciclo de Vida de Componentes

Ciclo de vida gerenciado para componentes da aplicação, permitindo que você defina fases de inicialização e finalização para seus serviços. Isso é útil para garantir que os recursos sejam corretamente inicializados e liberados.

Principais Funcionalidades

Configuração Simplificada

Configuração centralizada e simplificada de componentes, ajudando a evitar a propagação de parâmetros de configuração por todo o código.

Principais Funcionalidades

Integração com Bibliotecas Populares

Compatível com diversas bibliotecas populares no ecossistema Go, como `Uber's zap` para logging e prometheus para monitoramento.

Principais Funcionalidades

Testabilidade

A injeção de dependências e a modularidade facilitam a escrita de testes unitários e de integração, uma vez que as dependências podem ser facilmente mockadas.

Exemplo simples

Instância de Logger

```
// NewLogger cria uma nova instância de logger
func NewLogger() (*zap.Logger, error) {
    return zap.NewProduction()
}
```

Exemplo simples

Instância do Mux

```go
// NewMux cria um novo http.ServeMux
func NewMux(lc fx.Lifecycle, logger *zap.Logger) *http.ServeMux {
    mux := http.NewServeMux()
    lc.Append(fx.Hook{
        OnStart: func() error {
            logger.Info("Starting HTTP server")
            go http.ListenAndServe(":8080", mux)
            return nil
        },
        OnStop: func() error {
            logger.Info("Stopping HTTP server")
            return nil
        },
    })
    return mux
}
```

Exemplo simples

Registro de Handlers

```
// RegisterHandlers registra handlers HTTP
func RegisterHandlers(mux *http.ServeMux, logger *zap.Logger) {
    mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        logger.Info("Handling request", zap.String("path", r.URL.Path))
        w.Write([]byte("Hello, FX!"))
    })
}
```

Exemplo simples

Inicialização da aplicação

```
func main() {
    app := fx.New(
        fx.Provide(
            NewLogger,
            NewMux,
        ),
        fx.Invoke(RegisterHandlers),
    )
    app.Run()
}
```

Exemplo simples

`fx.Provide`

- **Função:** Registrar construtores de dependências no contêiner de dependências do `FX`.
- **Quando Usar:** Quando você precisa fornecer ou criar instâncias de dependências que serão usadas por outros componentes da aplicação.
- **Exemplo:** Logger, Repository, Service

Exemplo simples

`fx.Invoke`

- **Função:** Registrar funções que utilizam dependências para realizar tarefas ou iniciar processos. Essas funções são chamadas automaticamente durante a inicialização da aplicação.
- **Quando Usar:** Para registrar funções que precisam executar operações iniciais usando as dependências fornecidas, como configurar handlers, iniciar servidores, ou executar inicializações específicas.
- **Exemplo:** Handlers, Loggers, Middlewares, Conexão ao banco de dados, Arquivos de configurações da aplicação (yamls...)

Comparativo - Sem uso do FX

```go
func main() {
    // Configurar o logger
    logger := log.New(os.Stdout, "INFO: ", log.LstdFlags)

    // Configurar o servidor HTTP
    mux := http.NewServeMux()
    mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        logger.Println("Handling request:", r.URL.Path)
        w.Write([]byte("Hello, World!"))
    })

    // Iniciar o servidor HTTP
    server := &http.Server{
        Addr:    ":8080",
        Handler: mux,
    }

    go func() {
        logger.Println("Starting HTTP server on :8080")
        if err := server.ListenAndServe(); err != nil && err != http.ErrServerClosed {
            logger.Fatalf("Could not listen on :8080: %v\n", err)
        }
    }()

    if err := server.Close(); err != nil {
        logger.Fatalf("Server Close: %v", err)
    }

    logger.Println("Server gracefully stopped")
}
```

Comparativo - Com uso do FX

```go
func NewLogger() (*zap.Logger, error) {
    return zap.NewProduction()
}

func NewMux(lc fx.Lifecycle, logger *zap.Logger) *http.ServeMux {
    mux := http.NewServeMux()
    lc.Append(fx.Hook{
        OnStart: func() error {
            go http.ListenAndServe(":8080", mux)
            return nil
        },
        OnStop: func() error {
            return nil
        },
    })
    return mux
}

func RegisterHandlers(mux *http.ServeMux, logger *zap.Logger) {
    mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hello, FX!"))
    })
}

func main() {
    app := fx.New(
        fx.Provide(NewLogger, NewMux),
        fx.Invoke(RegisterHandlers),
    )
    app.Run()
}
```

Comparativo

Gerenciamento de Dependências

- **Sem FX:** As dependências são criadas e gerenciadas manualmente. Isso pode levar a um código mais acoplado e menos modular.
- **Com FX:** As dependências são declaradas e injetadas automaticamente, resultando em um código mais modular e desacoplado.

Comparativo

Ciclo de Vida dos Componentes

- **Sem FX:** O ciclo de vida (inicialização e finalização) dos componentes deve ser gerenciado manualmente, o que pode ser propenso a erros.
- **Com FX:** O ciclo de vida dos componentes é gerenciado automaticamente pelo FX, garantindo uma inicialização e finalização ordenadas e seguras.

Comparativo

Configuração e Manutenção

- **Sem FX:** A configuração é espalhada pelo código, o que pode dificultar a manutenção e a leitura.
- **Com FX:** A configuração é centralizada, facilitando a manutenção e a leitura do código.

Comparativo

Escalabilidade

- **Sem FX:** Adicionar novos componentes ou serviços pode requerer alterações significativas no código existente.
- **Com FX:** Adicionar novos componentes ou serviços é mais simples, pois o FX facilita a integração e a configuração de novos módulos.

"Concorrentes"

Wire (by Google)

- Gerador de código que cria automaticamente injeções de dependência para o seu aplicativo Go. Ele usa diretivas declarativas em seu código para gerar o código de injeção de dependências.

Dig (by Uber)

- Pacote de injeção de dependência baseado em contêiner, também desenvolvido pelo Uber, assim como o `FX`. É mais leve e focado apenas em injeção de dependências, sem o ciclo de vida dos componentes.

Dingo (by Open Table)

- Uma biblioteca de injeção de dependência para Go que é inspirada pelo Guice, a biblioteca de injeção de dependência para Java.

Exemplo de Estruturação

```
📦your-api
 ├ 📂 cmd
 |  └ 📂 api
 |  |  └ 📜 main.go
 └ 📂 internal
    └ 📂 domain
      └ 📂 handler
      |  ├ 📜 client_handler.go
      |  ├ 📜 tenant_handler.go
      |  └ 📜 module.go
      └ 📂 repository
      |  ├ 📜 client_repository.go
      |  ├ 📜 tenant_repository.go
      |  └ 📜 module.go
      └ 📂 service
        ├ 📜 client_service.go
        ├ 📜 tenant_service.go
        └ 📜 module.go
```