

Uma breve introdução ao service weaver

Componentes

Componentes são os principais constituintes do framework. Em código, um componente é representado como uma interface com anotações específicas do framework. Componentes podem conversar entre si por meio de chamadas de métodos. Em tempo de build, os componentes podem ser acoplados ou desacoplados entre si. Para um componente desacoplado, o Service Weaver gera um processo separado, que é capaz de se comunicar com os outros por gRPC. Dessa forma, o desenvolvedor tem total controle sobre a arquitetura do projeto, visto que cada componente pode tornar-se um microsserviço sem carga adicional de desenvolvimento.

Como exemplo, um componente cliente do mongodb:

```
package main

import (
    ...
)

type DbClient interface {
    Insert(context.Context, string, string, string) (string, error)
    ...
}

type dbClient struct {
    weaver.Implements[DbClient]
}

func (c *dbClient) Insert(context context.Context,
    database string, collection string, content string) (string, error) {

    client, err := c.getClient(context)
    log := c.Logger()
    if err != nil {
        log.Error("Falha ao obter cliente!", err)
    }
    client.Database(database).Collection(collection).InsertOne(context, content)
    return content, err
}
```

Além disso, é possível que componentes se comuniquem. Como exemplo, o serviço Reverser pode chamar o cliente mongo:

```
package service
...
type Reverser interface {
    Reverse(context.Context, string) (string, error)
}

type reverser struct {
    weaver.Implements[Reverser]
    mongo client.DbClient //componente de comunicação com o banco
}

func (r *reverser) Init(context.Context) error { // método "construtor" do Reverser
    r.mongo, err = weaver.Get[client.DbClient](r)
    return err
}

func (r *reverser) Reverse(c context.Context, s string) (string, error) {
    runes := []rune(s)
    n := len(runes)
    for i := 0; i < n/2; i++ {
        runes[i], runes[n-i-1] = runes[n-i-1], runes[i]
    }
    res, err := r.mongo.Insert(c, "Runas", "Invertidas", string(runes))
    ...
    return string(runes), nil
}
```

Preparando um projeto

Em cada pacote do projeto, é necessário executar os comandos:

```
go build .  
weaver generate .
```

Execução e monitoramento

Como um processo único

```
go run .  
weaver single dashboard
```

Como múltiplos processos

```
weaver multi deploy config.toml #arquivo de configuração do deploy  
weaver multi dashboard
```

Interface de monitoramento

Os comandos weaver dashboard iniciam um processo que permite acesso a uma tela com indicadores do deploy:

► Commands

▼ Components

Component	Replication	PIDs
client.DbClient	2	75729, 75742
service.Reverser	2	75707, 75718
main	2	75685, 75696

▼ Methods

Method	Count			Latency (ms)			Request (KB/s)			Reply (KB/s)		
	Min.	Hr.	All	Min.	Hr.	All	Min.	Hr.	All	Min.	Hr.	All
client.DbClient.Insert	0	7	7	0.0000	7.5918	7.5918	0.00	0.00	0.00	0.00	0.00	0.00
client.DbClient.StartConnection	0	0	0	0.0000	0.0000	0.0000	0.00	0.00	0.00	0.00	0.00	0.00
service.Reverser.Reverse	0	7	7	0.0000	7.9962	7.9962	0.00	0.00	0.00	0.00	0.00	0.00

A interface também oferece um gráfico de tráfego, que descreve a arquitetura do projeto:

▼ Traffic



Configuração do deploy

O framework permite a configuração do projeto por meio de um arquivo .toml, onde é possível determinar o comportamento dos componentes e variáveis de ambiente:

```
[serviceweaver]  
binary = "./weaver-study"  
env = ["MONGO_URI=mongodb://root:example@localhost:27017"]
```

Acoplando/desacoplando componentes

É possível acoplar ou separar componentes utilizando a variável *colocate* no arquivo de configuração. Por exemplo, a seguinte configuração acopla os componentes main e Reverser:

```
colocate = [  
    ["main", "github.com/mendoncas/weaver-study/service/Reverser"]  
]
```

O dashboard retrata a junção descrita na configuração:

▼ Traffic

