GO-11 03: Подтягиваем конфигурации из key-value store

Описание:

Давайте мы поговорим об очень полезной теме в создании любого приложения или сервиса - конфигурации. Для понимания того, зачем нам необходима конфигурация, давайте глянем на следующую проблему.
У нас есть сервис, который каким-либо образом связан с базой данных, например, с postgresql и для того, чтобы начать с ней работать, нам сначала требуется к ней подключиться.

```
func main() {
  db, err := sql.Open("postgres", "host=your_db port=5432 user=user
password=password dbname=production_db sslmode=disable")
  if err != nil {
    panic(err)
  }
  defer db.Close()

  // ... логика сервиса

  _ = http.ListenAndServe(":8080", nil)
}
```

А теперь представим ситуацию, когда у нас есть не одно окружение, а все по-взрослому (dev, test, stage, prod), и для каждого такого окружения нам нужно по-разному задавать параметры подключения к базе, порт и хост приложения, а также, возможно, и много других параметров.
Давайте посмотрим, как можно решить проблему при помощи конфигурации.
Конфигурация приложения - это, по сути, его настройка, мы как будто указываем параметры, с которыми хотим запустить наше приложение (например, строка подключения к базе, как в примере выше). Как другой пример, давайте вспомним о CLI утилитах, которыми мы пользуемся почти каждый день. Так вот, у многих CLI есть аргументы запуска, например, при отправке post запроса через утилиту curl мы часто указываем ей следующую конфигурацию:

```
curl -X POST localhost:8080/users -H 'Content-Type: application/json'
-d `{"name":"ivan"}`
```

- -X - параметр, который отвечает за метод запроса;

- -H - параметр, который добавляет к нашему запросу заголовок, например Content-Type;
- -d - параметр, который добавляет к нашему запросу body.

Таким же образом мы можем сконфигурировать и наше приложение, если научить его парсить аргументы командной строки, например:

```
./main -p 8080 -dsn "host=your_db port=5432 user=user
password=password dbname=production_db sslmode=disable"
```

И это уже будет считаться конфигурацией и имеет право на существование, но давайте лучше рассмотрим более надежные и красивые методы конфигурирования наших приложений.

Конфигурация внутри файла

Проблема подхода с аргументами командной строки состоит в том, что каждый запуск нашего приложения сопровождается огромной строкой с кучей аргументов, которые не всегда отражают то, к чему относятся, например, -p может быть как портом приложения, так и портом базы или другого сервиса. Также мы не можем удобно комбинировать и хранить наши аргументы (разве что сохранять команды запуска в отдельный bash скрипт, но это не выглядит как красивое решение и не решает все наши проблемы).

Гораздо более красивым способом будет хранить конфигурацию в отдельном json/yaml файле:

- Во-первых, это позволяет нам отделить логику приложения от конфигураций, которых может быть много.
- Во-вторых, мы можем организовать конфигурационный файл более гибко (благо, json/yaml имеют вложенность).
- В-третьих - хранение и переносимость, таким образом мы можем запустить наш бинарник, где угодно, имея при себе его конфигурационный файл.

Выглядеть такое решение может следующим образом:

```
// config.json
{
  "app": {
    "port": "8080"
  },
  "database": {
    "dialect": "postgres",
    "dsn": "host=your_db port=5432 user=user password=password
dbname=production_db sslmode=disable"
  }
}

type Config struct {
  App struct {
    Port: string `json:"port"`
```

```
  } `json:"app"`

  Database struct {
    Dialect string `json:"dialect"`
    DSN     string `json:"dsn"`
  } `json:"database"`
}

func main() {
  jsonFile, err := os.Open("config.json")
  if err != nil {
    fmt.Println(err)
  }
  defer jsonFile.Close()
  bytes, _ := ioutil.ReadAll(jsonFile)

  var config Config
  if err := json.Unmarshal(bytes, &config); err != nil {
    panic(err)
  }

  db, err := sql.Open(config.Database.Dialect, config.Database.DSN)
  if err != nil {
    panic(err)
  }
  defer db.Close()

  // ... логика сервиса

  _ = http.ListenAndServe(fmt.Sprintf(":%s", config.App.Port), nil)
}
```

Да кода стало немного больше, но зато мы получаем преимущества конфигурирования сервиса через файлы, а этот код можно спрятать в отдельный пакет.

Конфигурация в environment переменных

В отличие от файлов, которые нужно всегда носить рядом с приложением, конфигурация через переменные окружения может абстрагироваться от того, каким образом в окружении появляются параметры - они либо там есть, либо их нет.

Переменные окружения - это глобальные параметры системы, которые уникальны для одной конкретной сессии или процесса (объявив переменную окружения, а затем

перезапустив свою сессию, например, открыв новую вкладку в терминале, вы не увидите объявленных вами переменных).
В Go вы можете читать значения переменных окружения через стандартный пакет os.

```
func main() {
  db, err := sql.Open(os.Getenv("DIALECT"), os.Getenv("DSN"))
  if err != nil {
    panic(err)
  }
  defer db.Close()

  // ... логика сервиса

  _ = http.ListenAndServe(fmt.Sprintf(":%s", os.Getenv("PORT")), nil)
}
```

Либо используя такие библиотеки как envconfig и viper, которые позволяют очень гибко работать с переменными окружения и парсить их в go-структуру.
Вы скажете, что таким образом мы лишаемся возможности хранения параметров, и это отчасти так, но только отчасти. Ведь переменные окружения для сервисов чаще всего тоже помещаются в файл, такие файлы обычно называют dotenv или .env, и выглядят они примерно вот так:

```
APP_PORT=8080
DB_DIALECT=postgres
DB_DSN="host=your_db port=5432 user=user password=password
dbname=production_db sslmode=disable"
```

Они подгружаются различными способами непосредственно в процесс вашего приложения, например, вот так:

```
eval $(cat ./env | xargs -0) ./myapp
```

Самое главное, что приложению не важно, как в окружение попадут параметры, оно этим не управляет. И единственное, что оно может сделать, - это спаниковать из-за отсутствия какой-либо переменной.
Конфигурация в удаленном key-value хранилище
Теперь давайте рассмотрим вариант, когда у нас огромное количество сервисов и в каждом надо хранить конфигурацию, да еще и для нескольких окружений (dev/test/stage/prod). Управление таким количеством конфиг-файлов доставляет большие трудности и требует продуманного архитектурного подхода.
Одним из вариантов решения проблемы управления большим количеством конфиг-файлов является хранение их в удаленном key-value хранилище.
При таком подходе мы централизуем конфигурацию всех наших сервисов в одном месте и можем удобно ими управлять. Да, у такого подхода есть свои минусы, например,

отказоустойчивость такого решения, но эти проблемы решаются уже другими механизмами (кластеризацией, например).
Из примеров таких решений можно выделить:

- etcd
- Consul
- Zookeeper

Эти решения не всегда реализуют только key-value storage, но все они имеют этот функционал.
Мы будем рассматривать хранилище на примере Consul, хотя бы по той причине, что он написан на Go =)
Вообще, Consul - это универсальное средство для регистрации и настройки микросервисов, которое умеет

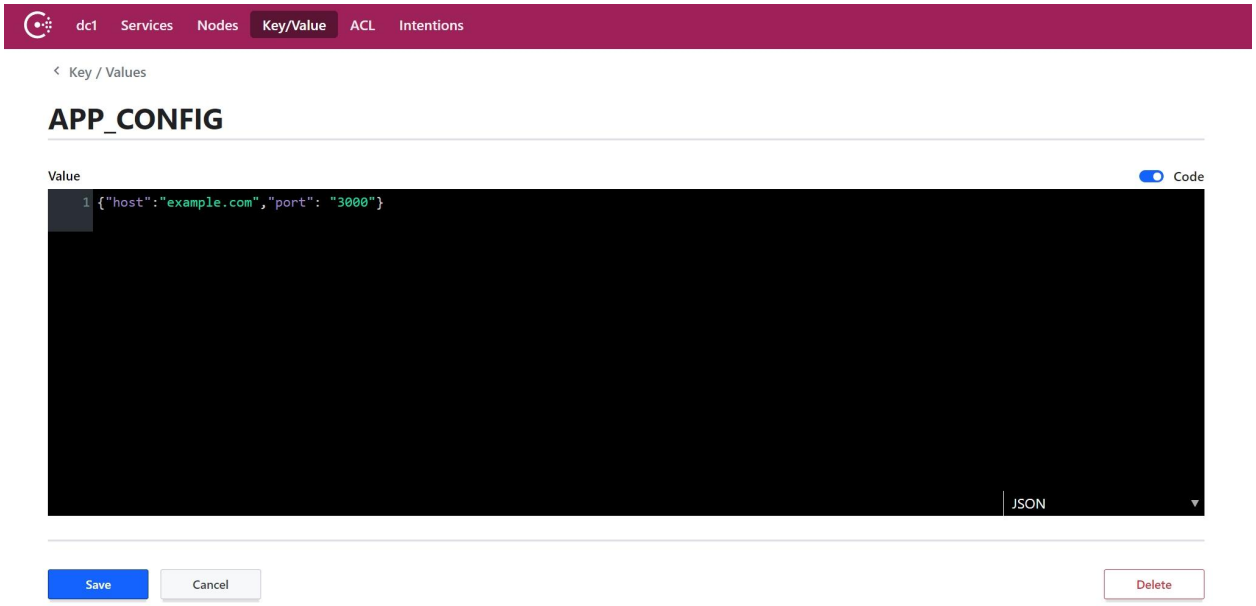
- обнаруживать и регистрировать сервисы - Service Discovery;
- проверять работоспособность сервисов - Health Check;
- исполнять роль Key-Value storage;
- обеспечивать безопасную связь между микросервисами Service Mesh.

Но сегодня мы остановимся именно на роли Key-Value Storage, которую умеет исполнять Consul. У Consul, так как он написан на go, есть нативное API. И для того чтобы работать с хранилищем, нам достаточно будет воспользоваться следующим примером.

```
package main

import (
  "fmt"
  "github.com/hashicorp/consul/api"
)

type Config struct {
  Port string `json:"port"`
}

func main() {
  // Get a new client
  client, err := api.NewClient(api.DefaultConfig())
  if err != nil {
    panic(err)
  }

  // Get a handle to the KV API
  kv := client.KV()

  // PUT a new KV pair
```

```
  p := &api.KVPair{Key: "APP_CONFIG", Value: []byte(`{"port":
"3000"}`)}
  _, err = kv.Put(p, nil)
  if err != nil {
    panic(err)
  }

  // Lookup the pair
  pair, _, err := kv.Get("APP_CONFIG", nil)
  if err != nil {
    panic(err)
  }

  var cfg Config
  if err := json.Unmarshal([]byte(pair.Value), &cfg); err != nil {
    panic(err)
  }
  fmt.Printf("App port: %s", cfg.Port)
}
```

Этот пример хорошо иллюстрирует, как мы можем писать данные в KV Consule storage и читать данные из него. Нам достаточно указывать уникальный ключ для нашего конфига (в нашем случае это APP_CONFIG), и при помощи методов put, get, delete, ...etc мы можем взаимодействовать с хранилищем. Но не только через библиотеку github.com/hashicorp/consul мы можем взаимодействовать с Consul, также мы можем обращаться к нему по HTTP API или посредством удобных библиотек для работы с конфигурациями в go, например, такой как viper.
Также после запуска этого кода мы можем зайти в UI Consul (если запускаете локально, тогда по дефолту ui будет висеть на localhost:8500) и посмотреть, что там действительно теперь хранится наша конфигурация.



Полезные ссылки:

- viper
- envconfig
- Consul lock
- Consul github
- Consul compose file
- Consul documentation
- Configuration management with Consul
- Configuration Management for Microservices
- Create a Distributed Semaphore with Consul Key-Value Store and Sessions

Задание:

В этом задании мы предлагаем вам познакомиться с хранилищем Consul самостоятельно, а также провзаимодействовать с ним при помощи удобной библиотеки viper или нативного клиента Consul.
Задание заключается в том, чтобы в нашем сервисе по ручке /config мы могли получить актуальное состояние конфига, которое может измениться без перезапуска сервиса.
Простыми словами - вам нужно сделать реалтайм конфиг для вашего сервиса.
Условия

- Должна быть ручка /config, которая отдает актуальное состояние конфига.
- При изменения конфига в Consul в сервисе конфиг также должен меняться.

- Можно использовать библиотеку для работы с конфигурациями viper, она облегчит работу с конфигурацией, или использовать нативный клиент Consul.
- Важно! Конфиг должен инициализироваться так, чтобы к нему можно было получить доступ откуда угодно (например, инициализировать в пакете main и передавать во все ручки при помощи middleware, или сделать всю инициализацию с пробросом, или сделать конфиг глобальной переменной, или любым другим способом, тут есть поле для вашего творчества).

Порядок действий

1. В вашем проекте module11 перейдите в новую ветку module11_03.
2. Запускаем Consul через make consul-up.
3. В пакете module11/internal/configs заполните логикой функцию GetConfig.
4. В пакете module11/internal/handlers/config написать логику ручки отдачи конфига в методе Handler.
5. В пакете module11/cmd/app зарегистрируйте ручку /config.

Если для реализации вашей идеи с доступностью конфигурации вам надо реорганизовать структуру ручек, это не запрещается!

6. Заведите нужный конфиг в Key-Value Consul любым удобным для вас способом (HTTP API, UI, Consul lib).
7. Проверьте работоспособность ручки /configs.
8. Попробуйте поменять вашу конфигурацию и, не перезагружая ваш сервис, снова вызвать ручку /config, конфигурация сервиса должна измениться.
9. В качестве ответа пришлите ссылку на merge request в ветку master вашего проекта ветки module11_03.