Если вы видите что-то необычное, просто сообщите мне.

Пишем REST API клиента на Go

Api клиенты, очень полезны когда вы выставляете ваш REST API на публику. И go делает это проще для вас, как для разработчика, так же как и для ваших пользователей, спасибо его структуре и типу систем. Но как определить хороший Api клиент?

В этой инструкции, мы собираемся обсудить несколько хороших практик написания SDK на Go.

Мы будем исполльзовть Facest.io Api как пример.

Прежде чем начнем писать какой-то код, мы одлжны изучить API чтобы понять главные его вещи, такие как:

- Что такое "Base url", и можно ли его поменять?
- Поддерживает ли он версионирование?
- Какие ошибки можно встретить?
- Каким образом аутентифицируются клиенты?

Ответы на эти вопросы помогут вам создать правильную структуру.

Начнем с основ. Создадим репозиторй, выберем название, в идеаеле одно должно совпадать с именем API сервиса. Инициализирум go модули. Создадим нашу главную структуру для хранения пользовательской информации. Это структура, в последствии, будет содержать точки доступа API как функции.

Структуры дложны быть гибкими, но так же ограниченными. чтобы пользователь не могу увидеть внутренние поля.

Создаем поля `BaseURL` и `HTTPClient` экспортируемыми, чтобы пользователь мог использовать их в своём HTTP клиенте, если это нужно.

```
package facest

import (
    "net/http"
    "time"
)

const (
    BaseURLV1 = "https://api.facest.io/v1"
)

type Client struct {
    BaseURL    string
    apiKey     string
    HTTPClient *http.Client
}

func NewClient(apiKey string) *Client {
    return &Client{
        BaseURL: BaseURLV1,
        apiKey:  apiKey,
        HTTPClient: &http.Client{
            Timeout: time.Minute,
        },
    }
}
```

Продолжаем: реализуем "Get Faces" точку доступа, которая возвращает список результатов и поддерживает постраничный вывод, что говорит о том, что постраничный вывод - это опция ввода.

Как я указал про API, любой ответ должен всегда иметь одну и ту же структуру, чтобы мы могли определить и отделить успешный ли был ответит или нет, от данных что пришли, чтобы пользователь видел только нужную ему информацию.

```
type errorResponse struct {
    Code    int    `json:"code"`
    Message string `json:"message"`
}

type successResponse struct {
    Code int         `json:"code"`
    Data interface{} `json:"data"`
}
```

Убедитесь, что вы не пишете все точки доступа в одном .go файле, но сгрупируйте их используя отдельные файлы, для примера вы можете сгрупировать их по типу, всё что начинается с `/v1/faces` идет в `fasec.go` файл.

Я обычно начинаю с определения типаов, вы можете это сделать вручную сконвертировав JSON в JSON-to-Go инструменте.

```
package facest

import "time"

type FacesList struct {
    Count      int    `json:"count"`
    PagesCount int    `json:"pages_count"`
    Faces      []Face `json:"faces"`
}

type Face struct {
    FaceToken  string      `json:"face_token"`
    FaceID     string      `json:"face_id"`
    FaceImages []FaceImage `json:"face_images"`
    CreatedAt  time.Time   `json:"created_at"`
}

type FaceImage struct {
    ImageToken string    `json:"image_token"`
    ImageURL   string    `json:"image_url"`
    CreatedAt  time.Time `json:"created_at"`
}
```

Функция `GetFaces` дожна поддерживать постраничный вывод а мы должны делать это добавив аргументы функции, но эти аргументы не обязательны и они могту быть изменены в будущем. Так, что стоит группировать их в специальную структуру:

```
type FacesListOptions struct {
    Limit int `json:"limit"`
    Page  int `json:"page"`
}
```

Еще один аргумент, нашей функции должен поддерживать, и его контекст, который позволит пользователю вызывать API. Пользователи могут создавать `Context`, передавая его в нашу функцию. Пример использования: отмена API вызова если он длится больше 5 секунд.

Теперь шаблон нашей функции выглядит следующим образом:

```
func (c *Client) GetFaces(ctx context.Context, options *FacesListOptions) (*FacesList, error)
{
    return nil, nil
}
```

Время сделать сам API:

```
func (c *Client) GetFaces(ctx context.Context, options *FacesListOptions) (*FacesList, error)
{
    limit := 100
    page := 1
    if options != nil {
        limit = options.Limit
        page = options.Page
    }

    req, err := http.NewRequest("GET", fmt.Sprintf("%s/faces?limit=%d&page=%d", c.BaseURL,
limit, page), nil)
    if err != nil {
        return nil, err
    }

    req = req.WithContext(ctx)
```

```
    res := FacesList{}
    if err := c.sendRequest(req, &res); err != nil {
        return nil, err
    }

    return &res, nil
}
func (c *Client) sendRequest(req *http.Request, v interface{}) error {
    req.Header.Set("Content-Type", "application/json; charset=utf-8")
    req.Header.Set("Accept", "application/json; charset=utf-8")
    req.Header.Set("Authorization", fmt.Sprintf("Bearer %s", c.apiKey))

    res, err := c.HTTPClient.Do(req)
    if err != nil {
        return err
    }

    defer res.Body.Close()

    if res.StatusCode < http.StatusOK || res.StatusCode >= http.StatusBadRequest {
        var errRes errorResponse
        if err = json.NewDecoder(res.Body).Decode(&errRes); err == nil {
            return errors.New(errRes.Message)
        }

        return fmt.Errorf("unknown error, status code: %d", res.StatusCode)
    }

    fullResponse := successResponse{
        Data: v,
    }
    if err = json.NewDecoder(res.Body).Decode(&fullResponse); err != nil {
        return err
    }

    return nil
}
```

Так как все точки доступа API работают одинаково, функция помошника `sendRequest` создана, чтобы избежать повторения в коде. Он задает большиство заголовков(content type, auth header), создает запрос, проверяет на ошибки, парсит ответ.

Отменит, что мы предполагаем ответы < 200 и >= 400, как ошибки, в этом случае парсится ответ из `errorResponse`. Понятно, что это зависит от архитектуры API, ваша Api может по разному обрабатывать ошибки.

Тестирование

Теперь у нас есть SDK с покрытием одной точки доступа API, что достаточно для этого примера, но этого достаточно чтобы передать это пользователю? Возможно да, но давайте посмотрим еще на несколько вещей.

Тесты востребованны в данном месте, и есть возможность двух типов: unit тесты, и интеграционные тесты. Для второго будем вызвать настоящий API. Напишем простой тест:

```
// +build integration

package facest

import (
    "os"
    "testing"

    "github.com/stretchr/testify/assert"
)

func TestGetFaces(t *testing.T) {
    c := NewClient(os.Getenv("FACEST_INTEGRATION_API_KEY"))

    ctx := context.Background()
    res, err := c.GetFaces(nil)

    assert.Nil(t, err, "expecting nil error")
    assert.NotNil(t, res, "expecting non-nil result")
```

```
    assert.Equal(t, 1, res.Count, "expecting 1 face found")
    assert.Equal(t, 1, res.PagesCount, "expecting 1 PAGE found")

    assert.Equal(t, "integration_face_id", res.Faces[0].FaceID, "expecting correct face_id")
    assert.NotEmpty(t, res.Faces[0].FaceToken, "expecting non-empty face_token")
    assert.Greater(t, len(res.Faces[0].FaceImages), 0, "expecting non-empty face_images")
}
```

Этот тест использует `env` переменную где указывается API ключ. Поступая таким образом, мы убеждаемся, что они не публичны. Позже мы можем сконфигурировать эту переменную с помощью окружения используемого в CI\CD.

Эти тесты, отьделены от юнит тестов(так как они выполняются гораздо дольше): execute):

```
go test -v -tags=integration
```

Документация.

Делайте ваш SDK не требующий описания с понятными типами и абстракциями. не выставляйте много информации. Обычно достаточно предоставить `godoc` ссылку как главную документацию.

Свместимость и версионирование.

Версионирование вашего SDK зависит от вашего репозитория. Но всегда убеждайтесь, что вы не сломали ничего в различных патчах или минорных выпусках. Обычно ваша SDK библиотека должна следовать API обновлениям, поэтому если релизитсся API v2, тогда должен быть и релиз SDK v2.

Revision #3
Created 2021-12-31 16:35:10 UTC by gasick
Updated 2023-04-16 19:30:03 UTC by gasick