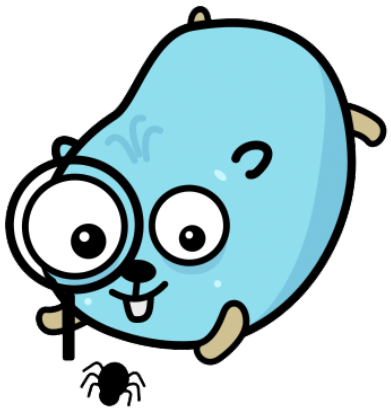
Работа с ошибками в тестах (часть 1)

В первой части модуля мы побудем в шкуре разработчика, которому необходимо написать и протестировать небольшую функцию `ParseToken`.



ParseToken: начало

Разработчику поступила задача написать функцию разбора и валидации JSON WEB токена (JWT). При этом сервис поддерживает единственный алгоритм подписи токена – `HS256`.

Примером JWT является

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg5MCIs
ImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJlZ
F9hdCI6MTYxMTY0MTkyNn0.IY-RsEAxCiDUch8_Rk77HvEDE8qhf1Zd3IUtxGlwd9w
```

Если раскрыть его, то получим

```
{
  "alg": "HS256",
  "typ": "JWT"
}
.
{
  "subject": "1234567890",
  "email": "john@gmail.com",
  "scopes": ["admin"],
```

```
  "expired_at": 1611641926
}
.

signature
```

Недолго думая, наш коллега написал функцию `ParseToken` (исходник примера).

Опустив детали, мы видим, что `ParseToken` имеет следующий алгоритм:

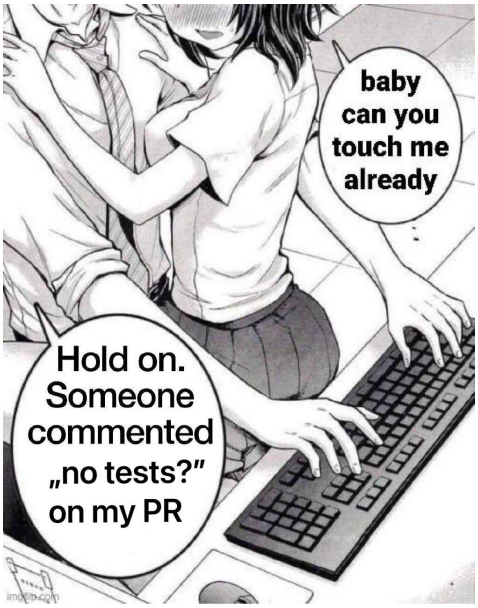
```
ParseToken:
    // Проверка входных аргументов.
    // Проверка формата токена.
    // Парсинг (parseHeader) и валидация заголовка токена.
    // Валидация подписи токена (verifySignature).
    // Парсинг (parsePayload) и валидация пайлоада.
    // Возврат данных из пайлоада (нижележащего токена) или ошибки.
```

Довольный своей работой разработчик отправил код на ревью.

Постарайтесь, не заглядывая в следующие шаги, обозначить для самого себя или написать в комментарии, какие недостатки имеет решение из примера выше :)

ParseToken: тесты: wantErr (bool)

На ревью поступил закономерный вопрос – "А где тесты?".



Наш коллега был знаком с табличными тестами и набросал подобные для всех возможных ветвлений в `ParseToken`.

Тест-кейсы решено было сделать бинарными относительно ошибки – просто ожидается она или нет (исходник примера):

```
package jwt

import (
    "reflect"
    "testing"
)

func TestParseToken(t *testing.T) {
    // Для составления тест-кейсов был использован https://jwt.io/
    // с алгоритмом HS256 и "secret" в качестве ключа хеширования.
    cases := []struct {
        name          string
        jwt           string
        wantErr       bool
        expectedToken Token
    }{
        {
            // {"alg": "HS256", "typ":  "JWT"}
```

```
            // {"subject": "1234567890", "email": "john@gmail.com",
"scopes": ["admin"], "expired_at": 4104586081}
            name:    "absolutely valid token",
            jwt:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg5MCI
sImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJl
ZF9hdCI6NDEwNDU4NjA4MX0.N7pFHBeew0mKBz4ULkim20QYbp7tcizR7Chdn4l32w8",
            wantErr: false,
            expectedToken: Token{
                Email:     "john@gmail.com",
                Subject:   "1234567890",
                Scopes:    []string{"admin"},
                ExpiredAt: 4104586081,
            },
        },
        {
            name:    "empty jwt",
            jwt:     "",
            wantErr: true,
        },
        {
            name:        "invalid token format",
            jwt:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg5MCI
sImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJl
ZF9hdCI6NDEwNDU4NjA4MX0",
            wantErrText: true,
        },
        {
            name:    "invalid header encoding",
            jwt:
"XXXXbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg5MCI
sImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJl
ZF9hdCI6NDEwNDU4NjA4MX0.N7pFHBeew0mKBz4ULkim20QYbp7tcizR7Chdn4l32w8",
            wantErr: true,
        },
        {
            name: "unsupported token type",
            // {"alg": "HS256", "typ":  "JWT123"}
            jwt:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVDEyMyJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg
5MCIsImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhw
aXJlZF9hdCI6NDEwNDU4NjA4MX0.6CM6Ys-eI1GUM0L0FG0d2yavG29FDujcklMc0ipNh
XA",
            wantErr: true,
```

```
        },
        {
            // {"alg": "HS512", "typ":  "JWT"}
            name:    "unsupported signing algo",
            jwt:
"eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVDEyMyJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg
5MCIsImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhw
aXJlZF9hdCI6NDEwNDU4NjA4MX0.ryWY506Gynnr-WhTZYBJiXLtKnSFRiPgXBRiFVU_P
JIvMwNAv0-p_dHcPF_x8JlwDxVqAYymogLFZqDFjRbp7w",
            wantErr: true,
        },
        {
            // signed by "secret123"
            name:    "invalid signature",
            jwt:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVDEyMyJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg
5MCIsImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhw
aXJlZF9hdCI6NDEwNDU4NjA4MX0.pjWE6Zr1UzHYqMKvlywOgeBhQMGSWt_1xEIV30H2j
xk",
            wantErr: true,
        },
        {
            name:    "invalid payload encoding",
            jwt:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.XXXdWJqZWN0IjoiMTIzNDU2Nzg5MCIs
ImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJlZ
F9hdCI6NDEwNDU4NjA4MX0.N7pFHBeew0mKBz4ULkim20QYbp7tcizR7Chdn4l32w8",
            wantErr: true,
        },
        {
            // 1611600481
            name:    "expired token",
            jwt:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg5MCI
sImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJl
ZF9hdCI6MTYxMTYwMDQ4MX0.MIVf6keNZGkoJPCajltFM7JNHPk6RXwmYxJbR_8_TE4",
            wantErr: true,
        },
    }
    for _, tt := range cases {
        t.Run(tt.name, func(t *testing.T) {
            token, err := ParseToken([]byte(tt.jwt),
[]byte("secret"))

            if tt.wantErr && err == nil {
```

```
                t.Fatalf("error was expected: got <nil>")
            }

            if !tt.wantErr && (err != nil) {
                t.Fatalf("unexpected error: %v", err)
            }

            if !reflect.DeepEqual(token, tt.expectedToken) {
                t.Fatalf("got token %#v, want: %#v", token,
tt.expectedToken)
            }
        })
    }
}
```

```
$ go test .
--- PASS: TestParseToken (0.00s)
    --- PASS: TestParseToken/absolutely_valid_token (0.00s)
    --- PASS: TestParseToken/empty_jwt (0.00s)
    --- PASS: TestParseToken/invalid_token_format (0.00s)
    --- PASS: TestParseToken/invalid_header_encoding (0.00s)
    --- PASS: TestParseToken/unsupported_token_type (0.00s)
    --- PASS: TestParseToken/unsupported_signing_algo (0.00s)
    --- PASS: TestParseToken/invalid_signature (0.00s)
    --- PASS: TestParseToken/invalid_payload_encoding (0.00s)
    --- PASS: TestParseToken/expired_token (0.00s)
ok      examples/06-working-with-errors-in-tests/parse-token-init
0.066s
```

Уже на этапе написания тестов разработчика посетило смутное сомнение – точно ли необходимое условие покрывает тот или иной тест. Замер тестового покрытия показал больше 90% . "Похоже на правду" – подумал наш друг и отправил код на очередную итерацию ревью.

P.S. В этом уроке в примерах и задачах специально не используется модуль **github.com/stretchr/testify**, т.к. мы будем отдельно касаться его чуть позже, а сейчас притворимся, что не были знакомы с ним.

Тест "Можно ли валидировать тест-кейсы покрытием?"

Выберите один вариант из списка

- ☐ Да
- ☐ Нет

ParseToken: тесты: expectedErrText

"Нет, так дело не пойдёт, – сказал тимлид. – Бинарные тесты подойдут для быстрого тестирования тривиальных функций, но это явно не наш случай".

Тимлиду хотелось, чтобы можно было однозначно соотнести тест и возвращаемую ошибку. Тестовое покрытие здесь не помощник, так как при неправильном составлении тест-кейсов какой-то тест может и не доходить до своего условия, а оно случайно покроется, например, другим тестом. В итоге какие-то тесты будут невалидные в принципе, а какие-то будут повторять друг друга, но мы об этом не узнаем.

Более того подобные тесты сложно понимать, потому что в случае их падения непонятно, что конкретно ожидалось от тестируемого кода:

```
if tt.wantErr && err == nil {
    t.Fatalf("error was expected: got <nil>") // Какая именно ошибка ожидалась?

}
```

Забыв про гошные заповеди, разработчик решил заменить `wantErr` на `expectedErrText`, ведь по тексту ошибки он точно сможет проверить, была ли протестирована нужная ветка в `ParseToken` (исходник примера):

```go
package jwt

import (
    "reflect"
    "strings"
    "testing"
)

func TestParseToken(t *testing.T) {
    // Для составления тест-кейсов был использован https://jwt.io/
    // с алгоритмом HS256 и "secret" в качестве ключа хеширования.
    cases := []struct {
        name            string
        jwt             string
        expectedErrText string
        expectedToken   Token
    }{
        {
            // {"alg": "HS256", "typ":  "JWT"}
            // {"subject": "1234567890", "email": "john@gmail.com",
"scopes": ["admin"], "expired_at": 4104586081}
            name:            "absolutely valid token",
            jwt:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg5MCI
sImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJl
ZF9hdCI6NDEwNDU4NjA4MX0.N7pFHBeew0mKBz4ULkim20QYbp7tcizR7Chdn4l32w8",
            expectedErrText: "",
            expectedToken: Token{
                Email:     "john@gmail.com",
                Subject:   "1234567890",
                Scopes:    []string{"admin"},
                ExpiredAt: 4104586081,
            },
        },
        {
            name:            "empty jwt",
            jwt:             "",
            expectedErrText: "empty jwt data",
        },
        {
            name:            "invalid token format",
            jwt:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg5MCI
sImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJl
ZF9hdCI6NDEwNDU4NjA4MX0",
```

```
            expectedErrText: "invalid token format",
        },
        {
            name:            "invalid header encoding",
            jwt:
"XXXXbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg5MCI
sImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJl
ZF9hdCI6NDEwNDU4NjA4MX0.N7pFHBeew0mKBz4ULkim20QYbp7tcizR7Chdn4l32w8",
            expectedErrText: "invalid character", // invalid
character ']' looking for beginning of value
        },
        {
            name: "unsupported token type",
            // {"alg": "HS256", "typ":  "JWT123"}
            jwt:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVDEyMyJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg
5MCIsImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhw
aXJlZF9hdCI6NDEwNDU4NjA4MX0.6CM6Ys-eI1GUM0L0FG0d2yavG29FDujcklMc0ipNh
XA",
            expectedErrText: "unsupported token type",
        },
        {
            // {"alg": "HS512", "typ":  "JWT"}
            name:            "unsupported signing algo",
            jwt:
"eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg5MCI
sImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJl
ZF9hdCI6NDEwNDU4NjA4MX0.YeYcZt06wTfXo62dD8qNvkhbersq_DySM3jmfS0SdeexT
CQPbDoqfzoV3HX023zufGIfuSpri5OnyYX39ABVxg",
            expectedErrText: "unsupported the signing algo",
        },
        {
            // signed by "secret123"
            name:            "invalid signature",
            jwt:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg5MCI
sImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJl
ZF9hdCI6NDEwNDU4NjA4MX0.EVWFGUfF7ZsJoz7TIXKV0SmJkc2VjYa6zniEIHDnPgk",
            expectedErrText: "invalid signature",
        },
        {
            name:            "invalid payload encoding",
            jwt:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.XXXdWJqZWN0IjoiMTIzNDU2Nzg5MCIs
```

```
ImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJlZ
F9hdCI6NDEwNDU4NjA4MX0.OB1AerP__4yOYkpvoV9GL6CKjra_IYSFLRfSEk6biw8",
            expectedErrText: "invalid character", // invalid
character ']' looking for beginning of value
        },
        {
            // 1611600481
            name:            "expired token",
            jwt:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg5MCI
sImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJl
ZF9hdCI6MTYxMTYwMDQ4MX0.MIVf6keNZGkoJPCajltFM7JNHPk6RXwmYxJbR_8_TE4",
            expectedErrText: "token was expired",
        },
    }
    for _, tt := range cases {
        t.Run(tt.name, func(t *testing.T) {
            token, err := ParseToken([]byte(tt.jwt),
[]byte("secret"))

            if tt.expectedErrText == "" && (err != nil) {
                t.Fatalf("unexpected error: %v", err)
            }

            if tt.expectedErrText != "" {
                if err == nil {
                    t.Fatalf("error with %q was expected: got <nil>",
tt.expectedErrText)
                }

                if !strings.Contains(err.Error(), tt.expectedErrText)
{
                    t.Fatalf("error with %q was expected, got %q",
tt.expectedErrText, err.Error())
                }
            }

            if !reflect.DeepEqual(token, tt.expectedToken) {
                t.Fatalf("got token %#v, want: %#v", token,
tt.expectedToken)
            }
        })
    }
}
```

С новыми тестами были найдены следующие ошибки в старых:

```
// Здесь на самом деле внутри "JWT123" и ParseToken возвращает другую ошибку.
{
    // {"alg": "HS512", "typ":  "JWT"}
    name:    "unsupported signing algo",
    jwt:
"eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVDEyMyJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg5MCIsImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJlZF9hdCI6NDEwNDU4NjA4MX0.ryWY506Gynnr-WhTZYBJiXLtKnSFRiPgXBRiFVU_PJIvMwNAv0-p_dHcPF_x8JlwDxVqAYymogLFZqDFjRbp7w",
    wantErr: true,

},

// Здесь аналогичная проблема. ParseToken даже не доходит до проверки подписи.
{
    // signed by "secret123"
    name:    "invalid signature",
    jwt:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVDEyMyJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg5MCIsImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJlZF9hdCI6NDEwNDU4NjA4MX0.pjWE6Zr1UzHYqMKvlywOgeBhQMGSWt_1xEIV30H2jxk",
    wantErr: true,

},

// Здесь неправильная подпись. До parsePayload функция не доходит.
{
    name:    "invalid payload encoding",
    jwt:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.XXXdWJqZWN0IjoiMTIzNDU2Nzg5MCIsImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJlZF9hdCI6NDEwNDU4NjA4MX0.N7pFHBeew0mKBz4ULkim20QYbp7tcizR7Chdn4l32w8",
    wantErr: true,

},
```

Довольный собой разработчик отправил код на очередное ревью.

Тест "О какой из заповедей забыл разработчик?"

Выберите один вариант из списка

- ☐ Don't communicate by sharing memory, share memory by communicating.
- ☐ Concurrency is not parallelism.
- ☐ Channels orchestrate; mutexes serialize.
- ☐ The bigger the interface, the weaker the abstraction.
- ☐ Make the zero value useful.
- ☐ interface{} says nothing.
- ☐ Gofmt's style is no one's favorite, yet gofmt is everyone's favorite.
- ☐ A little copying is better than a little dependency.
- ☐ Syscall must always be guarded with build tags.
- ☐ Errors are values.
- ☐ Cgo must always be guarded with build tags.
- ☐ Cgo is not Go.
- ☐ With the unsafe package there are no guarantees.
- ☐ Clear is better than clever.
- ☐ Reflection is never clear.
- ☐ Don't just check errors, handle them gracefully.
- ☐ Design the architecture, name the components, document the details.
- ☐ Documentation is for users.
- ☐ Don't panic.

ParseToken: errors are values

"Уже лучше", – подумал тимлид, но указал на следующие недостатки полученного решения.

1) Как мы помним, не следует работать с текстом ошибки – следует работать с её значением. Результат работы метода `Error` – это представление ошибки для людей, и в общем случае не стоит делать его частью контракта нашего API.

Подобные сравнения неусточивы: кто знает, как много ошибок могут содержать желаемый текст?

```
if !strings.Contains(err.Error(), tt.expectedErrText) { // Плохо!
    // ...

}
```

И даже точное сравнение не избавит нас от этой проблемы: что если нам вернули другую ошибку с аналогичным текстом? При этом ошибка может быть даже из другого пакета (например, IDE автоматически заимпортила нам не то, что нужно), а мы не узнаем об этом:

```
if !(err.Error() == tt.expectedErrText) { // Плохо!
    // ...

}
```

2) Подобные тесты нестабильны: ты поменял текст ошибки (например, исправил опечатку) и нужно менять тесты (хотя может кто-то увидит в этом плюс?).

3) Тесты, которые полагаются на внутренние/библиотечные ошибки в целом сложно поддерживать: приходится лезть в реализацию чужих функций; смотреть, какой текст содержат их ошибки; следить, не поменяется ли он со временем:

```
// Плохо!
{
    name:         "invalid header encoding",
    jwt:
"XXXXbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg5MCI
sImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJl
ZF9hdCI6NDEwNDU4NjA4MX0.N7pFHBeew0mKBz4ULkim20QYbp7tcizR7Chdn4l32w8",
    wantErrText: "invalid character", // invalid character ']'
looking for beginning of value

},
```

4) А как пользователям `ParseToken` различать возвращаемые ошибки? Ты же не будешь просить своих коллег полагаться на текст от метода `Error`?

Приняв замечания к сведению, разработчик сделал новую версию программы (исходник примера). Выделим основные моменты.

Появилось множество ошибок:

```
var (
    ErrEmptyJWT                = errors.New("empty jwt data")
    ErrInvalidTokenFormat      = errors.New("invalid token format:
'header.payload.signature' was expected")
    ErrInvalidHeaderEncoding   = errors.New("invalid header encoding")
    ErrUnsupportedTokenType    = errors.New("unsupported token type")
    ErrUnsupportedSigningAlgo = errors.New("unsupported the signing
algo")
    ErrInvalidSignature        = errors.New("invalid signature")
    ErrInvalidPayloadEncoding = errors.New("invalid payload
encoding")
    ErrExpiredToken            = errors.New("token was expired")

)
```

Где врапинг избыточен, ошибки возвращаются "как есть":

```
if len(jwt) == 0 {
    return Token{}, ErrEmptyJWT
}

parts := bytes.Split(jwt, byteDot)
if len(parts) != 3 {
    return Token{}, ErrInvalidTokenFormat
}

// ...

if time.Unix(t.ExpiredAt, 0).Before(time.Now()) {
    return Token{}, ErrExpiredToken

}
```

В остальных случаях оборачиваем в нашу ошибку чужую ошибку:

```
h, err := parseHeader(headerData)
if err != nil {
    return Token{}, fmt.Errorf("%w: %v", ErrInvalidHeaderEncoding,
err)
}

// ...

t, err := parsePayload(payloadData)
if err != nil {
    return Token{}, fmt.Errorf("%w: %v", ErrInvalidPayloadEncoding,
err)
}
```

Или обогащаем нашу ошибку дополнительным текстом:

```
if h.Typ != supportedTokenType {
    return Token{}, fmt.Errorf("%w: %q", ErrUnsupportedTokenType,
h.Typ)
}

if err := verifySignature(
    h.Alg,
    bytes.Join([][]byte{parts[0], parts[1]}, byteDot),
    signData,
    secret,
); err != nil {
    return Token{}, fmt.Errorf("verify signature: %w", err)
}

// ...

if algo != supportedAlgo {
    return fmt.Errorf("%w: %q", ErrUnsupportedSigningAlgo, algo)
}

// ...

if !hmac.Equal(sign, receivedSignature) {
    return fmt.Errorf("%w: %q vs expected %q", ErrInvalidSignature,
receivedSignature, sign)
}
```

Осталось дело за малым – поправить тесты!

ParseToken: тесты: errors.Is

Разработчик заменил

```
expectedErrText string
```

на

```
expectedErr error
```

и уже хотел было сравнивать ошибки через `==`, но вспомнил, что такое не пройдёт из-за использования `fmt.Errorf` и что нужно использовать забытый им `errors.Is` (исходник примера):

```
package jwt

import (
    "errors"
    "reflect"
    "testing"
)

func TestParseToken(t *testing.T) {
    // Для составления тест-кейсов был использован https://jwt.io/
    // с алгоритмом HS256 и "secret" в качестве ключа хеширования.
    cases := []struct {
        name          string
        jwt           string
        expectedErr   error
        expectedToken Token
    }{
        {
            // {"alg": "HS256", "typ":  "JWT"}
            // {"subject": "1234567890", "email": "john@gmail.com",
"scopes": ["admin"], "expired_at": 4104586081}
            name:        "absolutely valid token",
            jwt:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg5MCI
```

```
sImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJl
ZF9hdCI6NDEwNDU4NjA4MX0.N7pFHBeew0mKBz4ULkim20QYbp7tcizR7Chdn4l32w8",
            expectedErr: nil,
            expectedToken: Token{
                Email:     "john@gmail.com",
                Subject:   "1234567890",
                Scopes:    []string{"admin"},
                ExpiredAt: 4104586081,
            },
        },
        {
            name:        "empty jwt",
            jwt:         "",
            expectedErr: ErrEmptyJWT,
        },
        {
            name:        "invalid token format",
            jwt:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg5MCI
sImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJl
ZF9hdCI6NDEwNDU4NjA4MX0",
            expectedErr: ErrInvalidTokenFormat,
        },
        {
            name:        "invalid header encoding",
            jwt:
"XXXXbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg5MCI
sImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJl
ZF9hdCI6NDEwNDU4NjA4MX0.N7pFHBeew0mKBz4ULkim20QYbp7tcizR7Chdn4l32w8",
            expectedErr: ErrInvalidHeaderEncoding,
        },
        {
            name: "unsupported token type",
            // {"alg": "HS256", "typ":  "JWT123"}
            jwt:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVDEyMyJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg
5MCIsImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhw
aXJlZF9hdCI6NDEwNDU4NjA4MX0.6CM6Ys-eI1GUM0L0FG0d2yavG29FDujcklMc0ipNh
XA",
            expectedErr: ErrUnsupportedTokenType,
        },
        {
            // {"alg": "HS512", "typ":  "JWT"}
            name:        "unsupported signing algo",
```

```
            jwt:
"eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg5MCI
sImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJl
ZF9hdCI6NDEwNDU4NjA4MX0.YeYcZt06wTfXo62dD8qNvkhbersq_DySM3jmfS0SdeexT
CQPbDoqfzoV3HX023zufGIfuSpri5OnyYX39ABVxg",
            expectedErr: ErrUnsupportedSigningAlgo,
        },
        {
            // signed by "secret123"
            name:        "invalid signature",
            jwt:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg5MCI
sImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJl
ZF9hdCI6NDEwNDU4NjA4MX0.EVWFGUfF7ZsJoz7TIXKV0SmJkc2VjYa6zniEIHDnPgk",
            expectedErr: ErrInvalidSignature,
        },
        {
            name:        "invalid payload encoding",
            jwt:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.XXXdWJqZWN0IjoiMTIzNDU2Nzg5MCIs
ImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJlZ
F9hdCI6NDEwNDU4NjA4MX0.OB1AerP__4yOYkpvoV9GL6CKjra_IYSFLRfSEk6biw8",
            expectedErr: ErrInvalidPayloadEncoding,
        },
        {
            // 1611600481
            name:        "expired token",
            jwt:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWJqZWN0IjoiMTIzNDU2Nzg5MCI
sImVtYWlsIjoiam9obkBnbWFpbC5jb20iLCJzY29wZXMiOlsiYWRtaW4iXSwiZXhwaXJl
ZF9hdCI6MTYxMTYwMDQ4MX0.MIVf6keNZGkoJPCajltFM7JNHPk6RXwmYxJbR_8_TE4",
            expectedErr: ErrExpiredToken,
        },
    }
    for _, tt := range cases {
        t.Run(tt.name, func(t *testing.T) {
            token, err := ParseToken([]byte(tt.jwt),
[]byte("secret"))

            if !errors.Is(err, tt.expectedErr) {
                t.Fatalf("want error %v, got %v", tt.expectedErr,
err)
            }

            if !reflect.DeepEqual(token, tt.expectedToken) {
```

```
                t.Fatalf("got token %#v, want: %#v", token,
tt.expectedToken)
            }
        })
    }
}
```

Теперь не нужно отдельно обрабатывать `nil`-ошибку, т.к. `errors.Is` поддерживает это – кода стало гораздо меньше.

```
$ go test -v .
--- PASS: TestParseToken_v2 (0.00s)
    --- PASS: TestParseToken_v2/absolutely_valid_token (0.00s)
    --- PASS: TestParseToken_v2/empty_jwt (0.00s)
    --- PASS: TestParseToken_v2/invalid_token_format (0.00s)
    --- PASS: TestParseToken_v2/invalid_header_encoding (0.00s)
    --- PASS: TestParseToken_v2/unsupported_token_type (0.00s)
    --- PASS: TestParseToken_v2/unsupported_signing_algo (0.00s)
    --- PASS: TestParseToken_v2/invalid_signature (0.00s)
    --- PASS: TestParseToken_v2/invalid_payload_encoding (0.00s)
    --- PASS: TestParseToken_v2/expired_token (0.00s)

ok
examples/06-working-with-errors-in-tests/parse-token-with-sentinels
0.068s
```

Задача "ParseToken для безопасников"

Ссылка на заготовку.

Казалось бы, что работа над `ParseToken` закончена, но тут пришли ребята из ИБ и попросили собирать статистику, какие пользователи пытаются подсунуть нам невалидные токены (или кто-то пытается сделать это за них).

Вам требуется переписать `ParseToken` таким образом, чтобы, когда это возможно, она возвращала ошибку, из которой можно получить **email** пользователя, связанного с ней.

Все предыдущие тесты должны проходить, а также тимлид набросал вам новых, показывающих, как `ParseToken` будет использоваться в прикладном коде:

```
// ...

token, err := ParseToken([]byte(tt.jwt), []byte("secret"))

if !errors.Is(err, tt.expectedErr) {
    t.Fatalf("want error %q, got %q", tt.expectedErr, err)
}

if tt.expectedEmail != "" {
    var emailer interface {
        Email() string
    }
    if !errors.As(err, &emailer) {
        t.Fatalf("error %T doesn't implement `Email() string`
method", err)
    }

    if e.Email() != tt.expectedEmail {
        t.Fatalf("got email %#v, want: %#v", e.Email(),
tt.expectedEmail)
    }
}


// ...
```

- Принципиально сигнатуру функции менять нельзя:

```
func ParseToken(jwt, secret []byte) (Token, error)
```

- Все идентификаторы из известного `ParseToken` вам доступны (`byteDot`, `parsePayload`, `parseHeader`, `verifySignature`, ошибки и пр.) – определять их не нужно.
- Доступные импорты:

```
import (
    "bytes"
```

```
    "crypto/hmac"
    "crypto/sha256"
    "encoding/base64"
    "encoding/json"
    "errors"
    "fmt"
    "time"

)
```

- В тестах используются разные **JWT** с разными **email**.
- Можно и нужно определять новые типы.

Задача "Разделяй и властвуй"

[Ссылка на заготовку](#).

Вы могли заметить, что `parseHeader` и `parsePayload` реализованы так, что невозможно понять, какой именно этап кодировки токена выполнен неверно: перед нами невалидный **base64** или **JSON**?

```
func parseHeader(data []byte) (Header, error) {
    d := json.NewDecoder(
        base64.NewDecoder(base64.RawURLEncoding,
bytes.NewReader(data)),
    )

    var header Header
    return header, d.Decode(&header)

}

func parsePayload(data []byte) (Token, error) {
    d := json.NewDecoder(
        base64.NewDecoder(base64.RawURLEncoding,
bytes.NewReader(data)),
    )

    var token Token
    return token, d.Decode(&token)
}
```

В общем случае подобная детализация избыточна и не требуется, но мы предлагаем вам реализовать следующее.

К ошибкам невалидной кодировки были добавлены ещё две – `ErrInvalidBase64` и `ErrInvalidJSON`:

```
var (
    ErrInvalidHeaderEncoding  = errors.New("invalid header encoding")
    ErrInvalidPayloadEncoding = errors.New("invalid payload
encoding")

    ErrInvalidBase64 = errors.New("invalid base64")
    ErrInvalidJSON   = errors.New("invalid json")

)
```

Требуется переписать `parseHeader` и `parsePayload` таким образом, чтобы возможно было выполнять проверки вида:

```
h, err := parseHeader(headerData)

if errors.Is(err, ErrInvalidHeaderEncoding) {
    // Общий случай.
}

if errors.Is(err, ErrInvalidBase64) {
    // В заголовке невалидный base64.
}

if errors.Is(err, ErrInvalidJSON) {
    // В заголовке невалидный json.
}



t, err := parsePayload(payloadData)

if errors.Is(err, ErrInvalidPayloadEncoding) {
    // Общий случай.
}

if errors.Is(err, ErrInvalidBase64) {
    // В пайлоаде невалидный base64.
```

```
}

if errors.Is(err, ErrInvalidJSON) {
    // В пайлоаде невалидный json.
}
```

- Тестироваться будут только функции `parseHeader` и `parsePayload`.
- Принципиально менять сигнатуры функций нельзя:

```
func parseHeader(data []byte) (h Header, err error)

func parsePayload(data []byte) (t Token, err error)
```

- Типы `Header`, `Token` и ошибки выше вам доступны – их определять не нужно.
- Доступные импорты:

```
import (
    "bytes"
    "encoding/base64"
    "encoding/json"
    "errors"
    "fmt"
    "strings"

)
```

- Можно и нужно определять новые типы и функции.

Промежуточные выводы

На примере `ParseToken` мы рассмотрели организацию тестирования функций, возвращающих нетривиальные ошибки.

Выделим основные моменты:

- тесты могут быть как бинарными (есть ошибка / нет ошибки) `expectedErr bool`, так и ожидать конкретную ошибку `expectedErr error`;
- проверять ошибку необходимо с помощью `errors.Is` / `errors.As` и следует избегать проверки ошибки через её текст;
- выделение набора ошибок позволит вам быть уверенным, что вы протестировали именно те ветки условий, которые хотели;
- работа с ошибками без проблем укладывается в парадигму **табличных тестов**.

Во второй части урока мы подытожим полученные знания.