

Sistemas Embarcados

Aula 2 - Null Safety e Orientação à Objeto

```
// Import passport from 'passport'
import LocalStrategy from 'passport-local'
import { Strategy as JwtStrategy, ExtractJwt } from 'passport-jwt'
import passport from 'passport'

import User from '../models/user.model'
import constants from '../config/constants'
import { createError } from '../helpers/auth.helpers'

/**
 * Local Strategy Auth
 */
const localOpts = { usernameField: 'username' }

const localLogin = new LocalStrategy(
  localOpts,
  async (username, password, done) => {
    try {
      const user = await User.query().where('username', username)

      if (user.length === 0) {
        const userError = {
          username,
          password,
        }
        const createError = createError(userError)
        return done(null, createError)
      } else if (user[0].authenticateUser(password)) {
        return done(null, false)
      }
      return done(null, user[0])
    } catch (e) {
      return done(null, false)
    }
  }
)

// JWT Strategy Auth
const jwtOpts = {
  // Telling Passport to check authentication headers for JWT
  jwtFromRequest: ExtractJwt.fromAuthHeaderWithScheme('JWT'),
  // Telling Passport where to find the secret
  secretOrKey: constants.JWT_SECRET,
}

const jwtLogin = new JwtStrategy(jwtOpts, async (payload, done) => {
  try {
    console.log(payload)
    const user = await User.query().where('user_uuid', payload.user_uuid)
    console.log(user[0].toJSON())

    if (user.length === 0 || !user) {
      return done(null, false)
    }
    return done(null, user[0])
  }
})
```

Null Safety

Durante o desenvolvimento de software é comum ocorrerem erros durante a execução relacionado a valores `null`, ou seja, parte do código esperava receber um valor `int`, `bool`, `string`, etc, mas recebeu `null`. Este tipo de erro muitas vezes é difícil de rastrear e não são bem vistos por usuários de aplicativos.

Pensando neste problema, os criadores de linguagens mais modernas, como Dart, Kotlin, Swift e Rust, implementam a funcionalidade de Null Safety para transformar o problema de erro de execução para um erro estático em tempo de desenvolvimento.

Exemplo

```
// Without null safety:  
bool isEmpty(String texto) ⇒ texto.length == 0;  
  
main() {  
  isEmpty(null);  
}
```

```
// Using null safety:  
makeCoffee(String coffee, [String? dairy]) {  
  if (dairy != null) {  
    print('$coffee with $dairy');  
  } else {  
    print('Black $coffee');  
  }  
}
```

Princípios do Null Safety

- **Non-nullable by default:** A não ser que você diga ao Dart que uma variável pode receber null, ele considera sempre que todas as variáveis são `non-nullable`;
- **Incrementally adoptable:** Você escolhe o que migrar para *null safety* e quando. Você pode migrar incrementalmente, misturando código *null safety* com código *non null safety* no mesmo projeto;
- **Fully Sound:** O *null safety* é bastante robusto e auxilia durante o desenvolvimento de código. Se o sistema de tipagem identificar que uma variável não pode ser *null*, ela nunca será. Isto contribui não apenas para menos bugs, mas também para binários menores e execução mais rápida.

Operadores Null Safety

- **Nullable type (?):** utilizado para indicar que uma variável pode receber `null`. O operador deve ser inserido logo após o tipo da variável, por exemplo: `String?`;
- **Null assertion operator (!):** em alguns momentos o sistema de tipagem pode inferir que uma operação não poderá ser realizada porque uma das variáveis pode receber null. Porém, se você tem certeza que aquela variável nunca receberá `null` pode-se usar o operador `!` para reforçar pro sistema de tipagem que um valor não é `null` antes de realizar outra operação.

```
void main() {  
  int? a;  
  a = null;  
  print('a is $a.');
```

```
}  
  
void main() {  
  int a;  
  a = 42;  
  print('a is $a.');
```

Operadores condicionais

- **Acesso condicional de propriedades (?):** é possível verificar se uma variável é `null` antes de acessar suas propriedades. Por exemplo: `nome?.length`
- **Operador null-coalescing (??):** usado para atribuir um valor diferente quando uma variável possuir o valor `null`.

```
import "dart:io";

String? leNumero(){
  String? numero = stdin.readLineSync();
  return numero;
}

void main(){

  double dobro = double.parse(leNumero() ?? 0) * 2.0;
  print("O número em dobro é: $dobro");
}
```

Exercício da conversão de moeda

Paradigmas de Programação

O desenvolvimento de software pode ser realizado seguindo diferentes paradigmas. Em cada paradigma são definidas maneiras de estruturar o código e as melhores práticas ou padrões para se resolver os problemas aos quais o software será proposto. Cada linguagem de programação pode permitir o desenvolvimento de código em um ou mais paradigmas. A seguir estão alguns exemplos:

- Imperativo ou Procedural
- Declarativo
- Funcional
- Lógico
- Orientado à Eventos
- Orientado à objetos

Dart é uma linguagem orientada à objeto, com isso vamos entender melhor como funciona esse paradigma.

Orange Prune Rice

juice with cinnamon is a wonderful combination, especially with the prunes. It sounds sweet but it's more savoury than you might expect. You can serve it with a salad or other vegetables. I used broccoli.

Ingredients
Rice (Basmati) 300 ml
Carrots 1/2 Tbsp. oil
Cashews 300 ml
Prunes (dried plums) 1
Broccoli

Preparation
Dice onion.
Cut prunes into small pieces.
Cut carrots into slices.
Cut broccoli (white cooking rice).
Rinse rice under cold water.

Onions and carrots
medium/high heat
oil

Cinnamon, prunes and cashews
oil

Orange juice with stock
orange juice with stock so that you
double the amount of rice (so in
a 600 ml).

Boil
boil, then reduce heat to
and cover with a lid until liquid
is up and rice is done.

Cook Broccoli
Boil plenty of water,
then add broccoli stems
and some salt. After ~2
mins, add florets for
3-4 minutes. Then rinse
under cold water to
stop cooking process.



Orientação à Objeto

A programação orientada a objetos surgiu como uma alternativa às características da programação estruturada. O intuito da sua criação também foi o de aproximar o manuseio das estruturas de um programa ao manuseio das coisas do mundo real, daí o nome "objeto" como uma algo genérico, que pode representar qualquer coisa tangível.

Esse novo paradigma se baseia principalmente em dois conceitos chave: classes e objetos. Todos os outros conceitos, igualmente importantes, são construídos em cima desses dois.

Classes e objetos

As classes são como uma receita que vão definir as características e o comportamento do objeto. A classe define como o objeto deve ser construído, quais alterações são permitidas fazer no objeto, e quais operações o objeto pode realizar.

O objeto é uma instância de uma classe. Ou seja, é quando você dá vida a uma classe. Podemos comparar uma classe a uma planta baixa de uma casa, que tem todas as definições de como a casa deve ser construída e como ela vai funcionar, a casa em si será o objeto, a materialização do que foi descrito na planta baixa.



Codando uma classe

Classe

```
class Carro {  
    Double velocidade;  
    String modelo;  
  
    Carro(String modelo) {  
        this.modelo = modelo;  
        this.velocidade = 0;  
    }  
  
    void acelerar() {  
        /* código do carro para acelerar */  
    }  
  
    void frear() {  
        /* código do carro para frear */  
    }  
  
    void acenderFarol() {  
        /* código do carro para acender o farol */  
    }  
}
```

Objeto

```
import "Carro";  
  
void main(){  
  
    var carro = Carro("Onix");  
  
}
```

Exercícios

1. Desenvolva um programa em Dart para executar as operações de 3 figuras geométricas (quadrado, triângulo e círculo). Deverá ser desenvolvida uma classe para cada figura, e deverão ser criados métodos para calcular o comprimento da figura e sua área.

Solução