



## MESTRADO INFORMÁTICA APLICADA

### ARQUITETURA DE SOFTWARE

Prof. Fernando Bento  
Wilson Cristiano Oliveira

Nº180100286

[180100286@esg.ipstarem.pt](mailto:180100286@esg.ipstarem.pt)

Vasile Timotin

Nº 160173156

[160173015@esg.ipstarem.pt](mailto:160173015@esg.ipstarem.pt)

*Escola Superior de Gestão e Tecnologia de Santarém  
Complexo Andaluz, apartado 295, 2001-904 Santarém, Portugal*

# Microserviços

A arquitetura de microserviços é uma arquitetura de software que organiza uma aplicação como uma coleção de serviços pequenos e autônomos, cada um executando um processo de negócio específico e comunicando-se entre si através de APIs bem definidas.

Esta abordagem contrasta com a arquitetura monolítica tradicional, onde uma aplicação é construída como uma única unidade indivisível.

## Características Principais dos Microserviços:

1. **Autonomia:** Cada microserviço é independente e pode ser desenvolvido, implantado e escalado de forma separada dos outros. Isso facilita a manutenção e a atualização dos serviços sem impactar a aplicação como um todo.
2. **Descentralização:** Em vez de uma base de dados centralizada, cada microserviço pode ter sua própria base de dados, adequado às suas necessidades. Isso permite a escolha das tecnologias mais apropriadas para cada serviço.
3. **Comunicação via APIs:** Os microserviços se comunicam principalmente através de APIs HTTP/REST, mensageria ou outras formas de RPC (Remote Procedure Call). Essa comunicação é geralmente baseada em protocolos leves e padronizados.
4. **Implementação Independente:** Equipes diferentes podem desenvolver e implantar microserviços diferentes, permitindo que múltiplos processos de desenvolvimento ocorram em paralelo.
5. **Escalabilidade:** Os microserviços podem ser escalados de maneira independente. Se um serviço específico precisa de mais recursos, ele pode ser escalado sem a necessidade de escalar a aplicação inteira.

## Benefícios:

- **Flexibilidade:** Permite a adoção de diferentes tecnologias e linguagens de programação, conforme necessário para cada serviço.
- **Resiliência:** Falhas em um serviço específico não necessariamente afetam toda a aplicação, aumentando a robustez.
- **Manutenibilidade:** Facilita a identificação e correção de problemas, além de tornar o código mais modular e escalável.

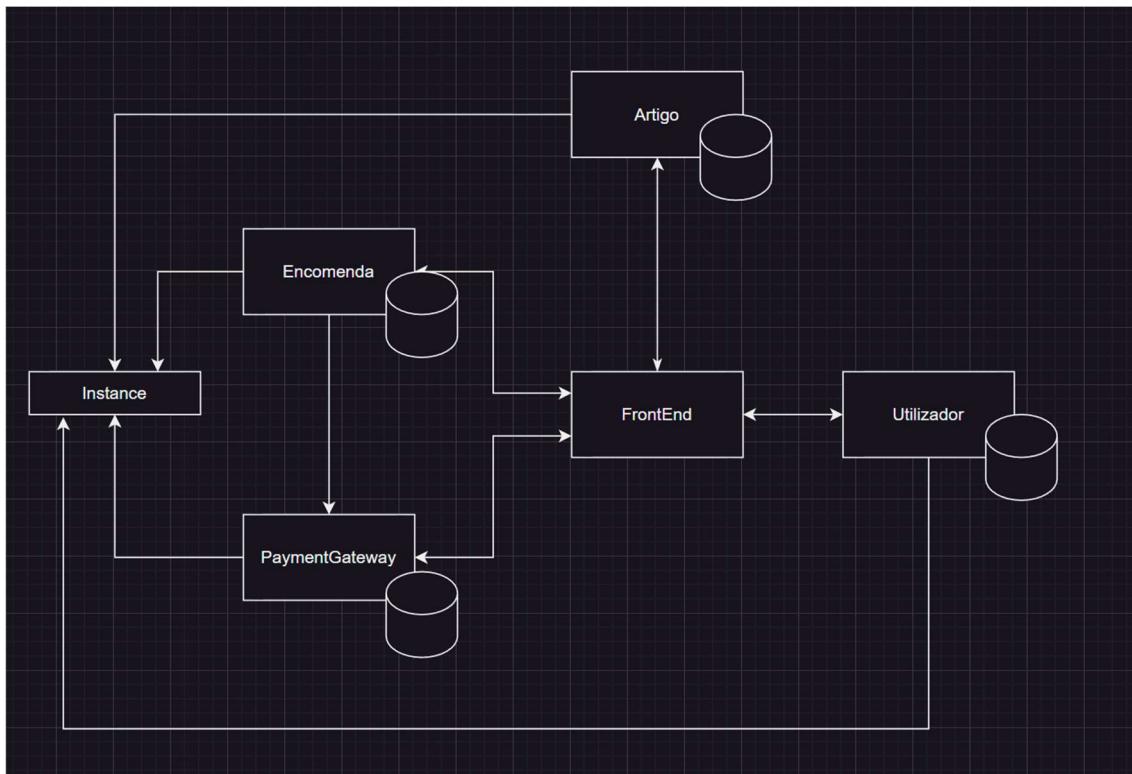
## Desafios:

- **Complexidade:** A gestão de múltiplos serviços independentes pode ser complexa, exigindo soluções robustas para orquestração, monitoramento e deployment.
- **Comunicação Inter-Serviços:** A comunicação entre serviços pode introduzir latência e falhas de rede, requerendo estratégias para garantir a consistência e a disponibilidade.
- **Teste e Debugging:** Testar e depurar uma aplicação distribuída é mais desafiador do que em uma aplicação monolítica.

## Arquitetura do nosso projeto

A arquitetura dos microserviços apresentada está organizada da seguinte forma, com cada microserviço:

1. **Artigo**
  - o Responsável pela gestão dos artigos disponíveis no sistema.
  - o Fornece serviços como criação, leitura, atualização e exclusão de artigos.
2. **Encomenda**
  - o Gere todo o ciclo de vida das encomendas.
  - o Inclui funcionalidades para criação de encomendas, atualização de status, rastreamento e cancelamento.
3. **FrontEnd**
  - o Representa a interface de usuário do sistema.
  - o Implementa a lógica de apresentação e interações do usuário com o sistema.
4. **Instance**
  - o Localização da instância da base de dados.
  - o Poderia ser utilizado para criar um monitoramento da aplicação.
5. **PaymentGateway**
  - o Responsável pela integração com gateways de pagamento.
  - o Gere transações financeiras, validação de pagamentos e retorno de status de pagamento.
6. **Utilizador**
  - o Gere os dados e a autenticação dos utilizadores.
  - o Fornece serviços de registo, login, atualização de perfil e gerenciamento de permissões.



## Representação das camadas de Microserviço:

Os nossos microserviços são compostos por um ficheiro app.py que é a Main da aplicação, ela é responsável por instanciar todo o microserviço utilizando um Model.

Temos uma pasta para a Base de Dados, utilizando a biblioteca SQLAlchemy juntamente com o Flask para disponibilizar os servidores em ambiente desenvolvimento Local.

Os serviços tem um config.py que serve de ficheiro de configuração do microserviço, de forma a melhorar a leitura e manutenção do código, contudo não foi muito aprofundada, e poderá ser melhorada.

Esse Modelo é a definição do contrato da aplicação, que por sua vez tem o acesso a BD, também como a definição da classe e dos seus métodos.

Routes.py é onde temos a definição de todas as rotas do microserviço que é a definição das nossas API REST, de forma a disponibilizar o acesso a informação da Base de dados aos nossos clientes autorizados.

A melhoria criada foi um método de healthcheck que serve para validar se o ambiente está ok e se todos os seus serviços estão no ar, de forma que este serviço é implementado na aplicação de forma de uma classe.

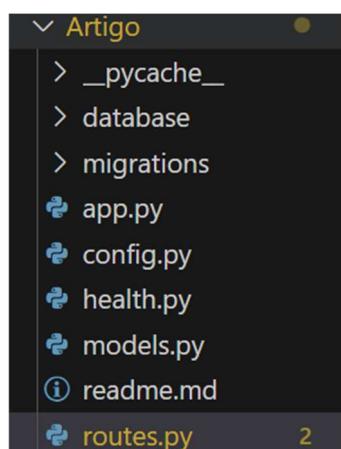
Por sua vez outra melhoria introduzida foi a documentação das API's através de swagger, que disponibiliza o JSON automaticamente com base no modelo e suas definições nas rotas.

Temos também um microserviço que faz de proxy para os serviços ou sistemas externos de pagamentos, que foi criado via Webservice Outsystems Application.

Todos os microserviços também contem um readme.md para descrever cada microserviço e como deverá ele ser instanciado e executado.

### 1. Artigo

Estrutura de pasta do microserviço:



## App.py:

```
Artigo > ℗ app.py > ...
1  from flask import Flask
2  from flask_migrate import Migrate
3  from flask_restx import Api
4  import models
5  import os
6  from routes import api as artigo_api_routes
7
8  app = Flask(__name__)
9  app.config['SECRET_KEY'] = 'j5sFMBkzzUV4DUTEQzxqFw'
10 app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
11
12 file_path = os.path.abspath(os.path.join(os.getcwd(), 'database', 'artigo.db'))
13 app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////' + file_path
14
15 models.init_app(app)
16
17 # Initialize Flask-Migrate
18 migrate = Migrate(app, models.db)
19
20 # Initialize Flask-RESTX
21 api = Api(app, title='Artigo API', doc='/swagger/', description='A microservice for artigo processing')
22 api.add_namespace(artigo_api_routes, path='/api/artigo')
23
24 if __name__ == '__main__':
25     app.run(debug=True, port=5002)
26
```

## Config.py:

```
Artigo > ℗ config.py > ...
1  import os
2  from apispec import APISpec
3  from apispec.ext.marshmallow import MarshmallowPlugin
4
5  basedir = os.path.abspath(os.path.dirname(__file__))
6
7  class Config:
8      SQLALCHEMY_DATABASE_URI = 'sqlite:///+' + os.path.join(basedir, 'database/artigo.db')
9      SQLALCHEMY_TRACK_MODIFICATIONS = False
10     APISPEC_SPEC = APISpec(
11         title='Artigo API',
12         version='v1',
13         plugins=[MarshmallowPlugin()],
14         openapi_version='2.0.0'
15     )
16     APISPEC_SWAGGER_UI_URL = '/swagger/'
17     APISPEC_SWAGGER_URL = '/swagger.json'
18
```

## Health.py:

```
Artigo > ℗ health.py > ...
1  from sqlalchemy.exc import OperationalError
2  from sqlalchemy.sql.expression import text
3  from models import db
4
5  class HealthCheck:
6      @classmethod
7          def check_database_status(cls):
8              try:
9                  # Execute a simple query to check the database status
10                 result = db.session.execute(text("SELECT 1"))
11                 # Check if the result is fetched successfully
12                 if result.scalar() == 1:
13                     return "OK"
14                 else:
15                     return "Error"
16             except OperationalError as e:
17                 print(f"Error checking database status: {e}")
18             return "Error"
19
```

## Models.py

```
Artigo > 🗂️ models.py > [o] db
  1  from flask_sqlalchemy import SQLAlchemy
  2  from flask_login import UserMixin
  3  from werkzeug.security import generate_password_hash
  4  from datetime import datetime, timezone
  5  ⚡
  6  db = SQLAlchemy()
  7
  8  def init_app(app):
  9      db.app = app
 10      db.init_app(app)
 11
 12  class Artigo(db.Model):
 13      id = db.Column(db.Integer, primary_key=True)
 14      descricao = db.Column(db.String(250), unique=False)
 15      codigoArtigo = db.Column(db.String(21), unique=True)
 16      preco = db.Column(db.Float, nullable=True)
 17      imagem = db.Column(db.String(255))
 18
 19      def __repr__(self):
 20          return f'<artigo {self.id}, {self.descricao}>'
 21
 22      def serializar(self):
 23          return {
 24              'id': self.id,
 25              'descricao': self.descricao,
 26              'codigoArtigo': self.codigoArtigo,
 27              'preco': self.preco,
 28              'imagem': self.imagem,
 29          }
 30
```

## Routes.py

```
Artigo > 🗂️ routes.py > ...
  1  from flask import request, jsonify
  2  from flask_restx import Namespace, Resource, fields
  3  from health import HealthCheck
  4  from models import Artigo, db
  5
  6  api = Namespace('artigo', doc='/swagger/', description='Artigo related operations')
  7  # Initialize Flask-RESTX
  8
  9  # Define the authorization header
 10  authorization = api.parser()
 11  authorization.add_argument('Authorization', location='headers', required=True, help='Bearer Token')
 12
 13  # Model for creating an article
 14  artigo_fields = api.model('Artigo', {
 15      'descricao': fields.String(required=True, description='Descrição do artigo'),
 16      'codigoArtigo': fields.String(required=True, description='Código do artigo'),
 17      'preco': fields.Float(required=False, description='Preço do artigo'),
 18      'imagem': fields.String(required=False, description='Imagem do artigo')
 19  })
 20  @api.route('/_health')
 21  class HealthCheckResource(Resource):
 22      def get(self):
 23          database_status = HealthCheck.check_database_status()
 24          if database_status == 'OK':
 25              return {'status': 'OK', 'database': 'OK'}, 200
 26          else:
 27              return {'status': 'Error', 'database': 'Error'}, 500
 28
 29  @api.route('/todos')
 30  class GetTodosArtigos(Resource):
 31      def get(self):
 32          todos_artigos = Artigo.query.all()
 33          result = [artigo.serializar() for artigo in todos_artigos]
 34          response = {
 35              'message': 'Todos os Artigos',
 36              'result': result
 37          }
 38          return jsonify(response)
 39
 40  @api.route('/criar')
 41  class CriarArtigo(Resource):
 42      @api.expect(authorization, artigo_fields)
 43      def post(self):
 44          try:
 45              args = authorization.parse_args()
 46              api_key = args['Authorization']
 47              if not api_key:
 48                  return {'message': 'Authorization header missing'}, 400
 49
 50              data = request.json
 51              descricao = data.get('descricao')
 52              codigoArtigo = data.get('codigoArtigo')
 53              preco = data.get('preco')
 54              imagem = data.get('imagem')
```

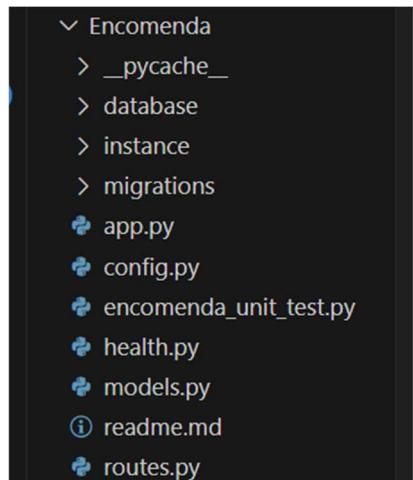
```

54     imagem = db.session.get(imagem)
55
56     if not descricao or not codigoArtigo:
57         response = {
58             'message': 'Descrição e CódigoArtigo são obrigatórios.'
59         }
60         return jsonify(response), 400
61
62     artigo = Artigo(
63         descricao=descricao,
64         codigoArtigo=codigoArtigo,
65         preco=preco,
66         imagem=imagem
67     )
68     db.session.add(artigo)
69     db.session.commit()
70     response = {
71         'message': 'Artigo criado com sucesso.',
72         'result': artigo.serializar()
73     }
74 except Exception as e:
75     print(str(e))
76     response = {'message': 'Erro na criação do artigo.'}
77     return jsonify(response), 500
78
79
80 @api.route('/<cA>')
81 class DetalhesArtigo(Resource):
82     def get(self, cA):
83         artigo = Artigo.query.filter_by(codigoArtigo=cA).first()
84         if artigo:
85             response = {'result': artigo.serializar()}
86         else:
87             response = {'message': 'Sem artigos criados.'}
88
89

```

## 2. Encomenda

Estrutura de pastas



## App.py

```
Encomenda > app.py > ...
1 import logging
2 import os
3 from flask import Flask
4 from flask_restx import Api
5 from flask_sqlalchemy import SQLAlchemy
6 from flask_migrate import Migrate
7 from models import db
8 import models
9 from routes import encomenda_blueprint
10 from routes import api as encomenda_api_routes
11
12 logging.basicConfig(level=logging.DEBUG)
13
14 app = Flask(__name__)
15 app.config['SECRET_KEY'] = 'j5sFMBkzzUV4DUTEQzxqFw'
16 app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
17
18 file_path = os.path.abspath(os.path.join(os.getcwd(), 'database', 'encomenda.db'))
19
20 app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///{} + file_path
21
22 models.init_app(app)
23
24 app.register_blueprint(encomenda_blueprint)
25
26 migrate = Migrate(app, db)
27
28 # Initialize Flask-RESTX
29 api = Api(app, title='Encomenda API', doc='/swagger/', description='A microservice for encomenda processing')
30 api.add_namespace(encomenda_api_routes, path='/api/encomenda')
31
32 if __name__ == '__main__':
33     logging.debug("Starting the application.")
34     app.run(debug=True, port=5003)
35
```

## Config.py

```
Encomenda > config.py > ...
● 1 ~ import os
2   from apispec import APISpec
3   from apispec.ext.marshmallow import MarshmallowPlugin
4
5   basedir = os.path.abspath(os.path.dirname(__file__))
6
7 ~ class Config:
8     SQLALCHEMY_DATABASE_URI = 'sqlite:///{} + os.path.join(basedir, 'database/encomenda.db')
9     SQLALCHEMY_TRACK_MODIFICATIONS = False
10 ~     APISPEC_SPEC = APISpec(
11         title='Encomenda API',
12         version='v1',
13         plugins=[MarshmallowPlugin()],
14         openapi_version='2.0.0'
15     )
16     APISPEC_SWAGGER_UI_URL = '/swagger/'
17     APISPEC_SWAGGER_URL = '/swagger.json'
18
```

## Health.py

```
Encomenda > health.py > ...
1  from sqlalchemy.exc import OperationalError
2  from sqlalchemy.sql.expression import text
3  from models import db
4
5  class HealthCheck:
6      @classmethod
7      def check_database_status(cls):
8          try:
9              # Execute a simple query to check the database status
10             result = db.session.execute(text("SELECT 1"))
11             # Check if the result is fetched successfully
12             if result.scalar() == 1:
13                 return "OK"
14             else:
15                 return "Error"
16         except OperationalError as e:
17             print(f"Error checking database status: {e}")
18             return "Error"
19
```

## Models.py

```
Encomenda > models.py > ...
1  from flask_sqlalchemy import SQLAlchemy
2
3  db = SQLAlchemy()
4
5  def init_app(app):
6      db.app = app
7      db.init_app(app)
8
9  class Encomenda(db.Model):
10     id = db.Column(db.Integer, primary_key=True)
11     utilizadorId = db.Column(db.Integer, nullable=False)
12     aberta = db.Column(db.Boolean, default=True)
13
14     def serializar(self):
15         return {
16             'id': self.id,
17             'utilizadorId': self.utilizadorId,
18             'aberta': self.aberta
19         }
20
21  class EncomendaLinha(db.Model):
22     id = db.Column(db.Integer, primary_key=True)
23     encomendaId = db.Column(db.Integer, db.ForeignKey('encomenda.id'), nullable=False)
24     artigoId = db.Column(db.Integer, nullable=False)
25     quantidade = db.Column(db.Integer, nullable=False)
26
27     def serializar(self):
28         return {
29             'id': self.id,
30             'encomendaId': self.encomendaId,
31             'artigoId': self.artigoId,
32             'quantidade': self.quantidade
33         }
```

## Routes.py

```
Encomenda > routes.py > ...
1  # routes.py
2  import logging
3  from flask import Blueprint, request, jsonify
4  import requests
5  from flask_restx import Namespace, Resource, fields
6  from health import HealthCheck
7  from models import Encomenda, EncomendaLinha, db
8  import logging
9
10 # Define the blueprint for encomenda routes
11 encomenda_blueprint = Blueprint('encomenda_api_routes', __name__, url_prefix='/api/encomenda')
12 api = Namespace("Encomenda", doc='/swagger/', description='Encomenda operations')
13
14 # Define authorization header
15 authorization = api.parser()
16 authorization.add_argument('Authorization', location='headers', required=True, help='Bearer Token')
17
18
19 # URL for the Utilizador API
20 UTILIZADOR_API_URL = 'http://127.0.0.1:5000/api/utilizador'
21
22 # Function to get user information from the Utilizador microservice
23 def get_utilizador(api_key):
24     headers = {'Authorization': api_key}
25     response = requests.get(UTILIZADOR_API_URL, headers=headers)
26     if response.status_code != 200:
27         return {'message': 'Não autorizado.'}
28     return response.json()
29
30 # Model for adding an article to the order
31 encomenda_fields = api.model('Encomenda', {
32     'artigoId': fields.Integer(required=True, description='ID do artigo'),
33     'quantidade': fields.Integer(required=True, description='Quantidade do artigo')
34 })
35
36 # Model for creating an order
37 criar_encomenda_fields = api.model('CriarEncomenda', {
38     'utilizadorId': fields.Integer(required=True, description='ID do utilizador')
39 })
40
41 @api.route('/_health')
42 class HealthCheckResource(Resource):
43     def get(self):
44         database_status = HealthCheck.check_database_status()
45         if database_status == 'OK':
46             return {'status': 'OK', 'database': 'OK'}, 200
47         else:
48             return {'status': 'Error', 'database': 'Error'}, 500
49
50
51 @api.route('/adicionarArtigo')
52 class AdicionarArtigo(Resource):
53     @api.expect(authorization, encomenda_fields)
54     def post(self):
55         try:
56             args = authorization.parse_args()
57             api_key = args['Authorization']
58             if not api_key:
59                 return {'message': 'Authorization header missing'}, 400
60
61             utilizador = get_utilizador(api_key)
62             if 'message' in utilizador:
63                 return utilizador, 401
64
65             data = request.json
66             if not all(key in data for key in ['artigoId', 'quantidade']):
67                 return {'message': 'Missing required parameters'}, 400
68
69             utilizador_id = utilizador.get('id')
70             if not utilizador_id:
71                 return {'message': 'User ID missing in response from Utilizador service'}, 400
72
73             encomenda = Encomenda.query.filter_by(utilizadorId=utilizador_id, aberta=True).first()
74             if not encomenda:
75                 encomenda = Encomenda(utilizadorId=utilizador_id, aberta=True)
76                 db.session.add(encomenda)
77                 db.session.commit()
78
79             linha_encomenda = EncomendaLinha(
80                 encomendaId=encomenda.id,
81                 artigoId=data['artigoId'],
82                 quantidade=data['quantidade']
83             )
84             db.session.add(linha_encomenda)
85             db.session.commit()
86
87             response = {
88                 'message': 'Artigo adicionado à encomenda com sucesso.',
89                 'result': linha_encomenda.serializar()
90             }
91         except Exception as e:
92             logging.error(f"Error adding article to order: {e}")
93             response = {'message': 'Erro ao adicionar artigo à encomenda.'}
94
95         return response, 200
```

```

96     @api.route('/criar')
97     class CriarEncomenda(Resource):
98         @api.expect(authorization, criar_encomenda_fields)
99         def post(self):
100             try:
101                 args = authorization.parse_args()
102                 api_key = args['Authorization']
103                 if not api_key:
104                     return {'message': 'Authorization header missing'}, 400
105
106                 utilizador = get_utilizador(api_key)
107                 if 'message' in utilizador:
108                     return utilizador, 401
109
110                 utilizador_id = utilizador.get('id')
111                 if not utilizador_id:
112                     return {'message': 'User ID missing in response from Utilizador service'}, 400
113
114                 encomenda = Encomenda(utilizadorId=utilizador_id, aberta=True)
115                 db.session.add(encomenda)
116                 db.session.commit()
117
118                 response = {
119                     'message': 'Encomenda criada com sucesso.',
120                     'result': encomenda.serializar()
121                 }
122             except Exception as e:
123                 logging.error(f"Error creating order: {e}")
124                 response = {'message': 'Erro na criação da encomenda.'}
125                 return response, 500
126
127             return response, 200
128
129     @api.route('/todos')
130     class GetTodosEncomendas(Resource):
131         def get(self):
132             try:
133                 todas_encomendas = Encomenda.query.all()
134                 response = [encomenda.serializar() for encomenda in todas_encomendas]
135                 return jsonify(response)
136             except Exception as e:
137                 logging.error(f"Error retrieving all orders: {e}")
138                 return {'message': 'Erro ao obter todas as encomendas.'}, 500
139

```

```

128     @api.route('/todos')
129     class GetTodosEncomendas(Resource):
130         def get(self):
131             try:
132                 todas_encomendas = Encomenda.query.all()
133                 response = [encomenda.serializar() for encomenda in todas_encomendas]
134                 return jsonify(response)
135             except Exception as e:
136                 logging.error(f"Error retrieving all orders: {e}")
137                 return {'message': 'Erro ao obter todas as encomendas.'}, 500
138
139     @api.route('/pendente')
140     class GetEncomendaPendente(Resource):
141         @api.expect(authorization)
142         def get(self):
143             try:
144                 args = authorization.parse_args()
145                 api_key = args['Authorization']
146                 if not api_key:
147                     return {'message': 'Authorization header missing'}, 400
148
149                 utilizador = get_utilizador(api_key)
150                 if 'message' in utilizador:
151                     return utilizador, 401
152
153                 utilizador_id = utilizador.get('id')
154                 if not utilizador_id:
155                     return {'message': 'User ID missing in response from Utilizador service'}, 400
156
157                 encomenda = Encomenda.query.filter_by(utilizadorId=utilizador_id, aberta=True).first()
158                 if encomenda:
159                     return encomenda.serializar()
160                 else:
161                     return {'message': 'No pending orders found for the user.'}, 404
162             except Exception as e:
163                 logging.error(f"Error retrieving pending order: {e}")
164                 return {'message': 'Erro ao obter a encomenda pendente.'}, 500
165

```

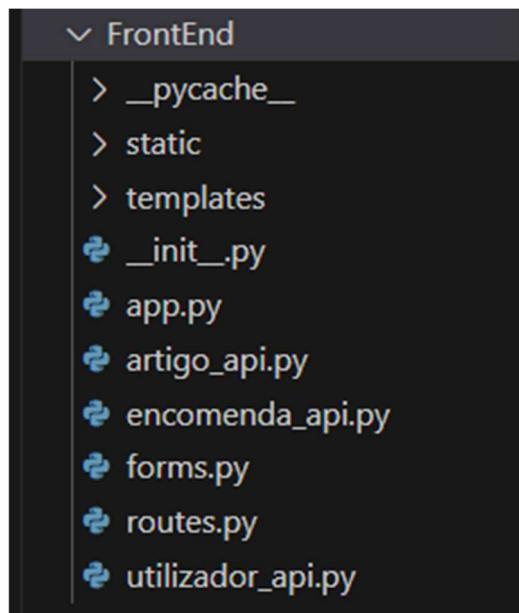
```

166     @api.route('/checkout')
167     class Checkout(Resource):
168         @api.expect(authorization)
169         def post(self):
170             try:
171                 args = authorization.parse_args()
172                 api_key = args['Authorization']
173                 if not api_key:
174                     return {'message': 'Authorization header missing'}, 400
175
176                 utilizador = get_utilizador(api_key)
177                 if 'message' in utilizador:
178                     return utilizador, 401
179
180                 utilizador_id = utilizador.get('id')
181                 if not utilizador_id:
182                     return {'message': 'User ID missing in response from Utilizador service'}, 400
183
184                 encomenda = Encomenda.query.filter_by(utilizadorId=utilizador_id, aberta=True).first()
185                 if not encomenda:
186                     return {'message': 'No open orders found for the user.'}, 404
187
188                 # Prepare payment data
189                 payment_data = {
190                     "Id": encomenda.id, # Payment ID should be unique
191                     "CustomerId": utilizador_id,
192                     "PaymentTypeId": 1, # Assuming a fixed payment type for now
193                     "TotalAmount": encomenda.total_amount, # Assuming `total_amount` is a property of Encomenda
194                     "Fee": encomenda.fee, # Assuming `fee` is a property of Encomenda
195                     "IsPaid": False
196                 }
197
198                 # Send request to Payment service
199                 payment_service_url = 'http://payment-service-url/payment/create' # Replace with actual URL
200                 response = requests.post(payment_service_url, json=payment_data)
201
202                 if response.status_code == 200:
203                     encomenda.aberta = False
204                     db.session.commit()
205                     return {'message': 'Checkout completed successfully and payment processed.'}, 200
206                 else:
207                     return {'message': 'Error processing payment', 'details': response.text}, 400
208
209             except Exception as e:
210                 logging.error(f"Error during checkout: {e}")
211             return {'message': 'Error no checkout.'}, 500

```

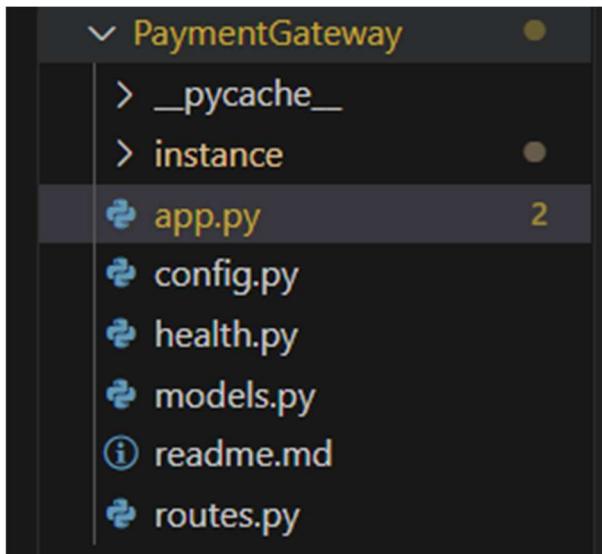
### 3. FrontEnd

Não vamos investir muito no Front End visto que foi utilizado, mas não era o foco do nosso projeto.



## 4. PaymentGateway

Estrutura do projeto:



App.py

```
PaymentGateway > app.py > ...
1 from flask import Flask
2 from flask_restx import Api
3 from routes import payment_ns
4 from models import db
5
6 # Initialize Flask app and configure
7 app = Flask(__name__)
8 app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///payments.db' # or another database URI
9 app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
10
11 # Initialize extensions
12 db.init_app(app)
13 api = Api(app, title='Payment API', doc='/swagger/', description='A microservice for payment processing')
14
15 # Register namespaces
16 api.add_namespace(payment_ns, path='/api/paymentGateway')
17
18 if __name__ == '__main__':
19     with app.app_context():
20         db.create_all() # Create database tables
21     app.run(debug=True, port=5001)
22
```

Config.py

```
PaymentGateway > config.py > ...
1 import os
2 from apispec import APISpec
3 from apispec.ext.marshmallow import MarshmallowPlugin
4
5 basedir = os.path.abspath(os.path.dirname(__file__))
6
7 class Config:
8     SQLALCHEMY_DATABASE_URI = 'sqlite:///{} + os.path.join(basedir, 'database/payments.db')
9     SQLALCHEMY_TRACK_MODIFICATIONS = False
10    APISPEC_SPEC = APISpec(
11        title='Payment API',
12        version='v1',
13        plugins=[MarshmallowPlugin()],
14        openapi_version='2.0.0'
15    )
16    APISPEC_SWAGGER_UI_URL = '/swagger/'
17    APISPEC_SWAGGER_URL = '/swagger.json'
18
```

## Health.py

```
PaymentGateway > health.py > ...
● 1  from sqlalchemy.exc import OperationalError
  2  from sqlalchemy.sql.expression import text
  3  from models import db
  4
 5  class HealthCheck:
 6      @classmethod
 7      def check_database_status(cls):
 8          try:
 9              # Execute a simple query to check the database status
10              result = db.session.execute(text("SELECT 1"))
11              # Check if the result is fetched successfully
12              if result.scalar() == 1:
13                  return "OK"
14              else:
15                  return "Error"
16          except OperationalError as e:
17              print(f"Error checking database status: {e}")
18          return "Error"
19
```

## Models.py

Esta classe define o modelo da base de dados interna do microserviço que serve de “cacher” dos pagamentos, sendo que é utilizada para guardar os pedidos e atualizar respostas com base na integração externa.

A integração externa é o serviço de Outsystems criado para este âmbito, tentar simular este processo de pagamento, serviço disponível publicamente.

<https://vasile-timotin.outsystemscloud.com/PaymentGateway/rest/Payment/>

Todos os serviços e chamadas para o serviço da payment gateway é feito na função do modelo de forma a integrar os dados na BD e no sistema externo.

```

PaymentGateway > models.py ...
1  from flask_sqlalchemy import SQLAlchemy
2  import requests
3  from sqlalchemy.sql.expression import text
4  # Define the database object
5  db = SQLAlchemy()
6
7  class Payment(db.Model):
8      id = db.Column(db.Integer, primary_key=True)
9      customer_id = db.Column(db.String(50), nullable=False)
10     payment_type_id = db.Column(db.Integer, nullable=False)
11     total_amount = db.Column(db.Float, nullable=False)
12     fee = db.Column(db.Float, nullable=True, default=0.0)
13     is_paid = db.Column(db.Boolean, default=False)
14
15     def __init__(self, customer_id, payment_type_id, total_amount, fee=0.0, is_paid=False, id=None):
16         if id is not None:
17             self.id = id
18         self.customer_id = customer_id
19         self.payment_type_id = payment_type_id
20         self.total_amount = total_amount
21         self.fee = fee
22         self.is_paid = is_paid
23
24     def to_dict(self):
25         return {
26             'id': self.id,
27             'customer_id': self.customer_id,
28             'payment_type_id': self.payment_type_id,
29             'total_amount': self.total_amount,
30             'fee': self.fee,
31             'is_paid': self.is_paid
32         }
33
34     @classmethod
35     def create_payment(cls, payment_data):
36         # Send data to the payment gateway
37         try:
38             response_data = create_payment_in_gateway(payment_data)
39             payment_id = response_data # The response is a string containing the payment ID
40             if not payment_id:
41                 raise Exception('Payment gateway did not return a payment ID.')
42             except Exception as e:
43                 raise Exception(f'Payment creation failed in gateway. Error: {e}')
44
45         # Store the payment in the local database with the returned ID
46         payment = cls(
47             id=payment_id,
48             customer_id=payment_data['CustomerId'],
49             payment_type_id=payment_data['PaymentTypeId'],
50             total_amount=payment_data['TotalAmount'],
51             fee=payment_data.get('Fee', 0.0),
52             is_paid=False
53         )
54         db.session.add(payment)
55         db.session.commit()
56         return payment
57
58     @classmethod
59     def update_payment_status(cls, payment_id, is_paid):
60         payment = cls.query.get(payment_id)
61         if payment:
62             payment.is_paid = is_paid
63             db.session.commit()
64             return payment
65         return None
66
67
68     @classmethod
69     def get_pending_payments(cls, filters=None):
70         query = cls.query.filter_by(is_paid=False) # Only pending payments
71
72         if filters:
73             if 'customer_id' in filters:
74                 query = query.filter_by(customer_id=filters['customer_id'])
75             if 'payment_type_id' in filters:
76                 query = query.filter_by(payment_type_id=filters['payment_type_id'])
77             if 'min_amount' in filters and filters['min_amount'] is not None:
78                 query = query.filter(cls.total_amount > float(filters['min_amount']))
79             if 'max_amount' in filters and filters['max_amount'] is not None:
80                 query = query.filter(cls.total_amount < float(filters['max_amount']))
81
82         return query.all()
83
84     # Function to integrate with the payment gateway for creating a payment
85     def create_payment_in_gateway(payment_data):
86         payment_gateway_url = 'https://vasile-timotin.outsystemscloud.com/PaymentGateway/rest/Payment/CreatePayment'
87         try:
88             response = requests.post(payment_gateway_url, json=payment_data)
89             response.raise_for_status()
90             return response.text # Return the string content of the response
91         except Exception as e:
92             print(f'Error creating payment in gateway: {e}')
93             raise
94
95     # Function to integrate with the payment gateway for acknowledging a payment
96     def acknowledge_payment_in_gateway(payment_id):
97         payment_gateway_url = 'https://vasile-timotin.outsystemscloud.com/PaymentGateway/rest/Payment/AcknolagePayment'
98         try:
99             response = requests.get(payment_gateway_url, params={'PaymentID': payment_id})
100            response.raise_for_status() # Raise an error for bad status codes
101            return response
102        except requests.RequestException as e:
103            print(f'Error acknowledging payment in gateway: {e}')
104            raise

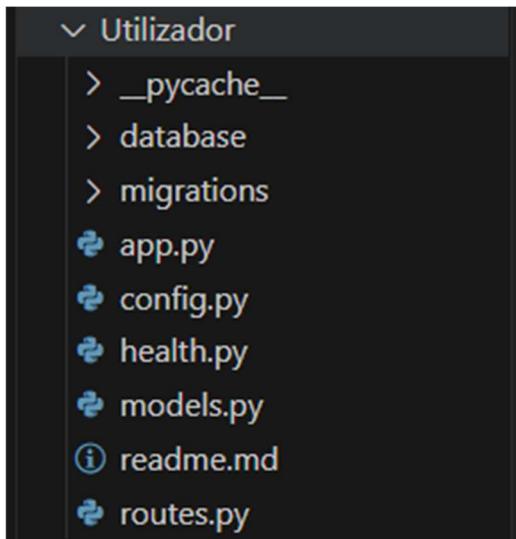
```

## Routes.py

```
1  # Inside routes.py
2
3  from flask import request
4  from flask_restx import Namespace, Resource, fields
5  from models import Payment, acknowledge_payment_in_gateway
6  from health import HealthCheck
7
8  #initialize payment namespace and services
9  payment_ns = Namespace('paymentGateway', description='Payment operations')
10
11 @payment_ns.model('Payment', {
12     'CustomerId': fields.String(required=True, description='The customer identifier'),
13     'PaymentTypeId': fields.Integer(required=True, description='The type of payment'),
14     'TotalAmount': fields.Float(required=True, description='The total amount of the payment'),
15     'Fee': fields.Float(required=False, description='The fee for the payment'),
16     'IsPaid': fields.Boolean(description='The payment status', default=False)
17 })
18
19 @payment_ns.route('/_health')
20 class HealthCheckResource(Resource):
21     def get(self):
22         database_status = HealthCheck.check_database_status()
23         if database_status == 'OK':
24             return {'status': 'OK', 'database': 'OK'}, 200
25         else:
26             return {'status': 'Error', 'database': 'Error'}, 500
27
28 @payment_ns.route('/CreatePayment')
29 class CreatePayment(Resource):
30     @payment_ns.expect(payment_model)
31     def post(self):
32         new_payment_data = payment_ns.payload
33         try:
34             payment = Payment.create_payment(new_payment_data)
35             return {'message': 'Payment created successfully.', 'payment': payment.to_dict()}, 201
36         except Exception as e:
37             return {'message': f'Error creating payment: {e}'}, 500
38
39 @payment_ns.route('/AcknowledgePayment')
40 class AcknowledgePayment(Resource):
41     @payment_ns.doc(params={'PaymentID': 'The payment identifier'})
42     def get(self):
43         payment_id = request.args.get('PaymentID')
44         if not payment_id:
45             return {'message': 'PaymentID parameter is required.'}, 400
46
47         payment = Payment.query.get(payment_id)
48         if not payment:
49             return {'message': 'Payment not found.'}, 404
50
51         if payment.is_paid:
52             return {'message': 'Payment is already acknowledged.'}, 400
53
54         try:
55             response = acknowledge_payment_in_gateway(payment_id)
56             payment.update_payment_status(payment_id, is_paid=True)
57             return {'message': 'Payment acknowledged.', 'payment': payment.to_dict()}, 200
58         except Exception as e:
59             return {'message': f'Error acknowledging payment: {e}'}, 500
60
61
62 @payment_ns.route('/PendingPayments')
63 class PendingPayments(Resource):
64     @payment_ns.doc(params={
65         'customerId': 'Filter by customer ID',
66         'paymentTypeId': 'Filter by payment type ID',
67         'minAmount': 'Filter by minimum amount',
68         'maxAmount': 'Filter by maximum amount'
69     })
70     def get(self):
71         filters = {
72             'customer_id': request.args.get('customerId'),
73             'payment_type_id': request.args.get('paymentTypeId'),
74             'min_amount': request.args.get('minAmount'),
75             'max_amount': request.args.get('maxAmount')
76         }
77
78         # Remove None values from filters
79         filters = {k: v for k, v in filters.items() if v is not None}
80
81         try:
82             pending_payments = Payment.get_pending_payments(filters)
83             return {'pending_payments': [payment.to_dict() for payment in pending_payments]}, 200
84         except ValueError as e:
85             return {'message': f'Invalid filter value: {e}'}, 400
```

## 5. Utilizador

Estrutura do microserviço:



App.py

```
Utilizador > app.py > load_user_from_request
1 from flask import Flask, g
2 from flask_sessions import SecureCookieSessionInterface # Add this import
3 from flask_migrate import Migrate
4 from flask_login import LoginManager
5 from flask_restx import Api
6 from models import db, init_app
7 import models
8 import os
9 from routes import utilizador_blueprint
10 from routes import api as utilizador_api_routes
11
12
13 app = Flask(__name__)
14 app.config['SECRET_KEY'] = 'j5sFMkzzUV4DUTEQzxqFw'
15 app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
16
17 file_path = os.path.abspath(os.path.join(os.getcwd(), 'database', 'utilizador.db'))
18
19 app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///{} + file_path
20
21 models.init_app(app)
22
23 app.register_blueprint(utilizador_blueprint)
24 login_manager = LoginManager(app)
25
26 migrate = Migrate(app, db)
27
28 # Initialize Flask-RESTX
29 api = Api(app, title='Utilizador API', doc='/swagger/', description='A microservice for user processing')
30 api.add_namespace(utilizador_api_routes, path='/api/utilizador')
31
32 @app.route('/')
33 def index():
34     return "Bem vindo as API do Utilizador"
35
36 @login_manager.request_loader
37 def load_user_from_request(request):
38     api_key = request.headers.get('Authorization')
39     if api_key:
40         api_key = api_key.replace('Basic ', '', 1)
41         utilizador = models.Utilizador.query.filter_by(api_key=api_key).first()
42         if utilizador:
43             return utilizador
44     return None
45
46 class CustomSessionInterface(SecureCookieSessionInterface):
47     """ Impedir a criação de sessões a partir de solicitações de API """
48     def save_sessions(self, *args, **kwargs):
49         if g.get('login_via_header'):
50             return
51         return super(CustomSessionInterface, self).save_session(*args, **kwargs)
52
53 if __name__ == '__main__':
54     app.run(debug=True, port=5000)
```

## Config.py

```
Utilizador > config.py > ...
1 import os
2 from apispec import APISpec
3 from apispec.ext.marshmallow import MarshmallowPlugin
4
5 basedir = os.path.abspath(os.path.dirname(__file__))
6
7 class Config:
8     SQLALCHEMY_DATABASE_URI = 'sqlite:///+' + os.path.join(basedir, 'database/utilizador.db')
9     SQLALCHEMY_TRACK_MODIFICATIONS = False
10    APISPEC_SPEC = APISpec(
11        title='Utilizador API',
12        version='v1',
13        plugins=[MarshmallowPlugin()],
14        openapi_version='2.0.0'
15    )
16    APISPEC_SWAGGER_UI_URL = '/swagger/'
17    APISPEC_SWAGGER_URL = '/swagger.json'
18
```

## Health.py

```
Utilizador > health.py > ...
1 from sqlalchemy.exc import OperationalError
2 from sqlalchemy.sql.expression import text
3 from models import db
4
5 class HealthCheck:
6     @classmethod
7     def check_database_status(cls):
8         try:
9             # Execute a simple query to check the database status
10            result = db.session.execute(text("SELECT 1"))
11            # Check if the result is fetched successfully
12            if result.scalar() == 1:
13                return "OK"
14            else:
15                return "Error"
16        except OperationalError as e:
17            print(f"Error checking database status: {e}")
18        return "Error"
19
```

## Models.py

```
Utilizador > models.py > ...  
1  from flask_sqlalchemy import SQLAlchemy  
2  from flask_login import UserMixin  
3  from werkzeug.security import generate_password_hash  
4  from datetime import datetime, timezone  
5  
6  db = SQLAlchemy()  
7  
8  def init_app(app):  
9      db.app = app  
10     db.init_app(app)  
11  
12 class Utilizador(db.Model, UserMixin):  
13     id = db.Column(db.Integer, primary_key=True)  
14     nomeUtilizador = db.Column(db.String(50), unique=False)  
15     password = db.Column(db.String(250), unique=True)  
16     administrador = db.Column(db.Boolean)  
17     api_key = db.Column(db.String(255), unique=True, nullable=True)  
18     ativo = db.Column(db.Boolean, default=True)  
19     autenticado = db.Column(db.Boolean, default=False)  
20  
21     def __repr__(self):  
22         return f'Utilizador {self.id}, {self.nomeUtilizador}'  
23  
24     def serializar(self):  
25         return {  
26             'id': self.id,  
27             'nomeUtilizador': self.nomeUtilizador,  
28             'administrador': self.administrador,  
29             'api_key': self.api_key,  
30             'ativo': self.ativo,  
31         }  
32  
33     def update_api_key(self):  
34         self.api_key = generate_password_hash(self.nomeUtilizador + str(datetime.now(timezone.utc)))  
35
```

## Routes.py

```
Utilizador > routes.py > CurrentUtilizador > get  
1  import datetime  
2  from urllib import request  
3  from flask import Blueprint, jsonify, request, make_response  
4  from werkzeug.security import generate_password_hash, check_password_hash  
5  from flask_restx import Namespace, Resource, fields  
6  from models import Utilizador, db  
7  from flask_login import login_user, current_user, logout_user  
8  from health import HealthCheck  
9  
10  ##var  
11  utilizador_blueprint = Blueprint('utilizador_api_routes', __name__, url_prefix='/api/utilizador')  
12  api = Namespace('Utilizador', doc='/swagger/', description='Utilizador operations')  
13  
14  # Define models for request and response payloads  
15  utilizador_model = api.model('Utilizador', {  
16      'nomeUtilizador': fields.String(required=True, description='Nome do utilizador'),  
17      'password': fields.String(required=True, description='Senha do utilizador')  
18  })  
19  #healthcheck  
20  @api.route('/health')  
21  class HealthCheckResource(Resource):  
22      def get(self):  
23          database_status = HealthCheck.check_database_status()  
24          if database_status == 'OK':  
25              return {'status': 'OK', 'database': 'OK'}, 200  
26          else:  
27              return {'status': 'Error', 'database': 'Error'}, 500  
28  
29  #get all  
30  @api.route('/')  
31  class UtilizadorList(Resource):  
32      @api.doc(responses={200: 'Success', 500: 'Internal Server Error'})  
33      def get(self):  
34          """List all utilizadores"""  
35          todos_utilizadores = Utilizador.query.all()  
36          result = [utilizador.serializar() for utilizador in todos_utilizadores]  
37          return jsonify({'message': 'Todos os Utilizadores', 'result': result})  
38  
39  @api.doc(responses={201: 'Created', 400: 'Bad Request', 500: 'Internal Server Error'})  
40  @api.expect(utilizador_model)  
41  def post(self):  
42      """Create a new utilizador"""  
43      data = request.json  
44      nomeUtilizador = data.get('nomeUtilizador')  
45      password = data.get('password')  
46  
47      if not nomeUtilizador or not password:  
48          return {'message': 'Nome de utilizador e senha são obrigatórios.'}, 400  
49  
50      utilizador = Utilizador(nomeUtilizador=nomeUtilizador, password=generate_password_hash(password))  
51      db.session.add(utilizador)  
52      db.session.commit()  
53  
54      return {'message': 'Utilizador criado com sucesso.', 'result': utilizador.serializar()}, 201  
55
```

```

56     @api.route('/<string:nomeUtilizador>/login')
57     class UtilizadorLogin(Resource):
58         @api.doc(responses={200: 'Success', 401: 'Unauthorized', 500: 'Internal Server Error'})
59         @api.expect(utilizador_model)
60         def post(self, nomeUtilizador):
61             """Login a utilizador"""
62             data = request.json
63             password = data.get('password')
64
65             utilizador = Utilizador.query.filter_by(nomeUtilizador=nomeUtilizador).first()
66
67             if not utilizador or not check_password_hash(utilizador.password, password):
68                 return {'message': 'Autenticação incorreta'}, 401
69
70             login_user(utilizador)
71             return {'message': 'Conectado','result':utilizador.serializar()}, 200
72
73     @api.route('/logout')
74     class UtilizadorLogout(Resource):
75         @api.doc(responses={200: 'Success', 401: 'Unauthorized', 500: 'Internal Server Error'})
76         def post(self):
77             """Logout current utilizador"""
78             if current_user.is_authenticated:
79                 logout_user()
80                 return {'message': 'Desconectado'}, 200
81             return {'message': 'Não existem utilizadores conectados'}, 401
82
83     @api.route('/<string:nomeUtilizador>/existe')
84     class UtilizadorExist(Resource):
85         @api.doc(responses={200: 'Success', 404: 'Not Found', 500: 'Internal Server Error'})
86         def get(self, nomeUtilizador):
87             """Check if utilizador exists"""
88             utilizador = Utilizador.query.filter_by(nomeUtilizador=nomeUtilizador).first()
89             if utilizador:
90                 return {'message': True}, 200
91             return {'message': False}, 404
92
93     @api.route('/current')
94     class CurrentUtilizador(Resource):
95         @api.doc(responses={200: 'Success', 401: 'Unauthorized', 500: 'Internal Server Error'})
96         def get(self):
97             """Get current utilizador"""
98             if current_user.is_authenticated:
99                 return {'result': current_user.serializar()}, 200
100            else:
101                return {'message': 'Utilizador não conectado'}, 401
102
103    @api.route('/<string:nomeUtilizador>/update-api-key')
104    class UpdateAPITKey(Resource):
105        @api.doc(responses={200: 'Success', 401: 'Unauthorized', 404: 'Not Found', 500: 'Internal Server Error'})
106        def post(self, nomeUtilizador):
107            """Update API Key for utilizador"""
108            utilizador = Utilizador.query.filter_by(nomeUtilizador=nomeUtilizador).first()
109            if utilizador:
110                utilizador.update_api_key()
111                db.session.commit()
112                login_user(utilizador)
113                return {'message': 'Conectado', 'api_key': utilizador.api_key}, 200
114            return {'message': 'Utilizador não encontrado'}, 404
115

```

Documentação automática de cada serviço com o seu swagger, dando liberdade ao utilizador de testar diretamente as API's no swagger, executando assim o Curl que está por trás.

**Utilizador API 1.0**

[ Base URL: / ]  
/swagger.json

A microservice for user processing

**Utilizador** Utilizador operations

**GET** /api/utilizador/ List all utilizadores

Parameters

No parameters

Responses

Code Description

200	Success
500	Internal Server Error

Response content type application/json

**POST** /api/utilizador/ Create a new utilizador

**GET** /api/utilizador/\_health

**GET** /api/utilizador/current Get current utilizador

**POST** /api/utilizador/logout Logout current utilizador

**GET** /api/utilizador/{nomeUtilizador}/existe Check if utilizador exists

**POST** /api/utilizador/{nomeUtilizador}/login Login a utilizador

**POST** /api/utilizador/{nomeUtilizador}/update-api-key Update API Key for utilizador

**Models**

```
Utilizador <-
  nomeUtilizador* > [...]
  password* > [...]
}
```

**Payment API 1.0**

[ Base URL: / ]  
/swagger.json

A microservice for payment processing

**paymentGateway** Payment operations

**GET** /api/paymentGateway/AcknowledgePayment

**POST** /api/paymentGateway/CreatePayment

Parameters

Name Description

**payload** \* required

object  
(body)

```
{
  "CustomerId": "string",
  "PaymentTypeId": 0,
  "Type": "string",
  "Fee": 0,
  "IsPaid": false
}
```

Example Value | Model

Parameter content type application/json

Responses

Code Description

200	Success
-----	---------

Response content type application/json

**GET** /api/paymentGateway/PendingPayments

**GET** /api/paymentGateway/\_health

**Models**

**Encomenda API** 1.0

[ Base URL: / ]  
/swagger.json

A microservice for encomenda processing

### Encomenda

Encomenda operations

**POST** /api/encomenda/adicionarArtigo

Parameters

Name	Description
Authorization * required	Bearer Token string (header)
payload * required	Example Value   Model object (body)

Try it out

Responses

Code	Description
200	Success

**POST** /api/encomenda/checkout

**POST** /api/encomenda/criar

**GET** /api/encomenda/pendente

**GET** /api/encomenda/todos

---

Models

**Artigo API** 1.0

[ Base URL: / ]  
/swagger.json

A microservice for artigo processing

### artigo

Artigo related operations

**POST** /api/artigo/criar

Parameters

No parameters
---------------

Responses

Code	Description
200	Success

Response content type application/json

**GET** /api/artigo/todos

**GET** /api/artigo/{cA}

Models

Por sua vez cada swagger tem um swagger.json que facilmente é feito o download de um ficheiro JSON que é importado em ferramentas como o Postman.

Na imagem abaixo vamos observar um ficheiro e como fica importado no Postman.

```

{
  "swagger": "2.0",
  "info": {
    "version": "1.0"
  },
  "paths": {
    "/api/utilizador/": {
      "get": {
        "responses": {
          "200": {
            "description": "Success"
          },
          "500": {
            "description": "Internal Server Error"
          }
        },
        "summary": "List all utilizadores",
        "operationId": "get_utilizador_list"
      },
      "post": {
        "responses": {
          "201": {
            "description": "Created"
          },
          "400": {
            "description": "Bad Request"
          },
          "500": {
            "description": "Internal Server Error"
          }
        },
        "summary": "Create a new utilizador",
        "operationId": "post_utilizador_list"
      }
    },
    "/api/utilizador/_health": {
      "get": {
        "responses": {
          "200": {
            "description": "Success"
          }
        },
        "summary": "Get health check resource",
        "operationId": "get_health_check_resource"
      }
    },
    "/api/utilizador/current": {
      "get": {
        "responses": {
          "200": {
            "description": "Success"
          },
          "401": {
            "description": "Unauthorized"
          },
          "500": {
            "description": "Internal Server Error"
          }
        },
        "summary": "Get current utilizador",
        "operationId": "get_current_utilizador"
      }
    },
    "/api/utilizador/logout": {
      "post": {
        "responses": {
          "200": {
            "description": "Success"
          },
          "401": {
            "description": "Unauthorized"
          }
        }
      }
    }
  }
}

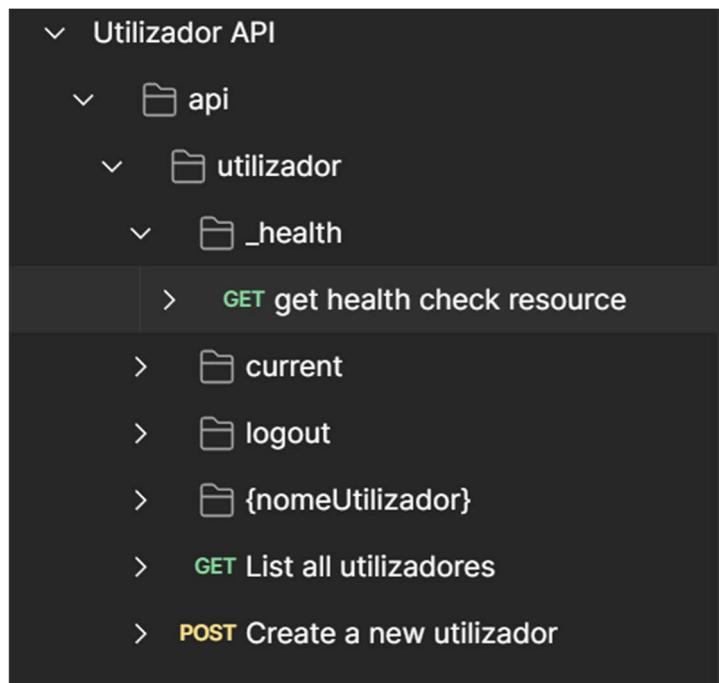
```

Importação no Postman:

The Postman interface shows the following sections:

- Workspaces:** Shows 'Big\_Data\_mongo' and 'gRPC Product Management'.
- Top Bar:** Includes 'New', 'Import' (highlighted by a red arrow), and several API endpoint buttons: POST CreateProduct, PUT UpdateProduct, GET GetProducts, GET GetProductById, POST findOne, and Release Notes.
- Central Area:** Displays the 'Postman v11.2.0' header and a 'What's New' section with a note about forkable collections.
- Import Dialog:** An overlay titled 'SpaceCash Payments' shows the 'Choose how to import your API' section. It has two options: 'Postman Collection' (selected) and 'OpenAPI 2.0 with a Postman Collection'. A red arrow points to the 'Import' button at the bottom right of this dialog.
- Bug Fixes:** A list of fixes for version 11.1.14, including issues with RAML 1.0 definitions and vault secrets.
- Postman v11.1.14:** A summary of security updates and bug fixes.

Resultado:



Utilização do healthcheck do serviço Utilizador.

HTTP Utilizador API / api / utilizador / \_health / get health check resource

Send

Params Authorization Headers (6) Body Scripts Tests Settings Cookies

Query Params

Key	Value	Description	...	Bulk Edit
Key	Value	Description	...	Bulk Edit

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

Status: 200 OK Time: 35 ms Size: 210 B Save as example

```
1 {  
2   "status": "OK",  
3   "database": "OK"  
4 }
```

## Conclusão Aplicação do GP:

Podemos então concluir que a arquitetura de microserviços é um modelo de sistemas distribuídos, sendo que a decentralização destes módulos dá vantagens para a extensibilidade do sistema em questão, na sua manutenção e evolução de código.

Contudo há maiores dificuldades de debugging dos fluxos end to end do sistema, sendo que os erros podem ocorrer em sistemas diferentes, sendo que é importantíssimo um sistema de logging e alertas que devem ajudar nesse trabalho.

Toda a comunicação entre os nossos serviços utilizam uma interoperabilidade utilizando integrações REST, sendo utilizado JSON para a comunicação entre serviços ou interfaces API.

Poderíamos utilizar RPC ou outro qualquer mecanismo de comunicação entre os sistemas, sendo que a grande vantagem disto é que cada sistema poderá ser desenhado e desenvolvido com base na sua funcionalidade e com a tecnologia necessária.

No nosso exemplo a aplicação outsystems comunica com os nossos serviços que estão feitos utilizando o flask, também como as Bases de dados são ou podem ser diferentes.

Concluimos também que cada microserviço tem um ciclo de vida próprio, e não dependem uns dos outros para fazerem o seu processamento, contudo se algum serviço parar, poderemos ter impactos em certas funcionalidades do sistema.