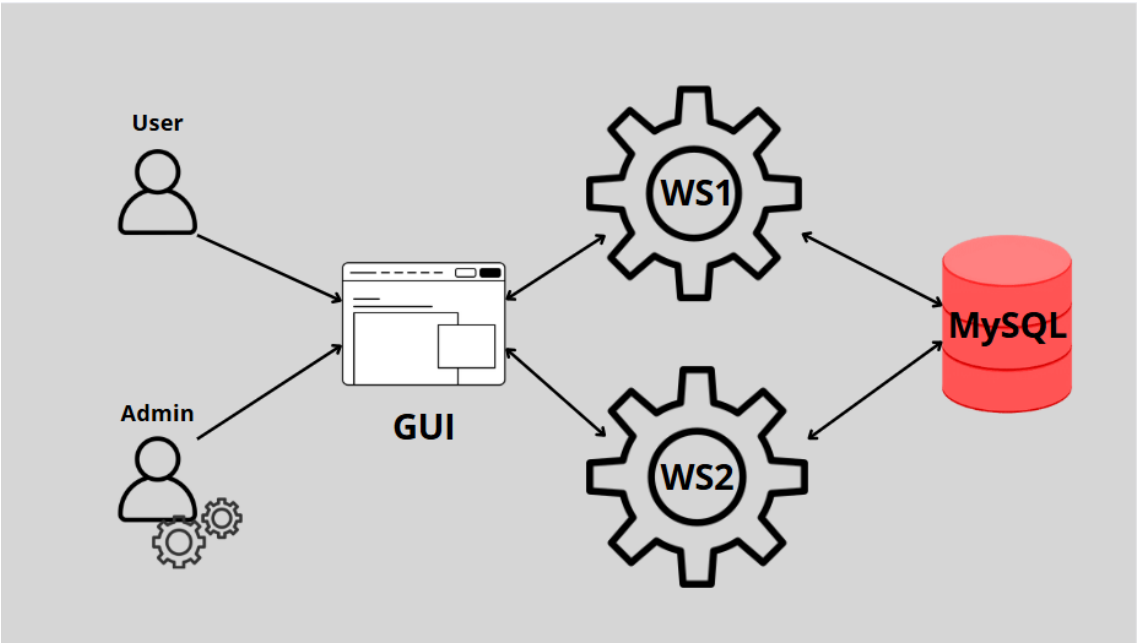
Unidade Curricular de Sistemas Distribuídos e Segurança

2º ano da Licenciatura de Tecnologias Digitais e Segurança de Informação

Atividade Laboratorial - XML based Communication and Web Services



Alexandre Sorochenko | Nº123530 | Tecnologias Digitais e Segurança de Informação | 24/25

André Santos Camponez Baiona | Nº123527 | Tecnologias Digitais e Segurança de Informação | 24/25

Jorge Henrique Pessoa Dias | Nº123535 | Tecnologias Digitais e Segurança de Informação | 24/25

Docente: João Pedro Pavia

Maio, 2025

Índice

1. Objetivo

O objetivo deste laboratório é desenvolver e demonstrar uma aplicação web distribuída ***Online Tracking System*** composta por uma interface gráfica e dois serviços web SOAP que disponibilizam operações para utilizadores e administradores, aplicando os princípios da arquitetura orientada a serviços (SOA).

2. Descrição do Problema e Requisito

É pedido para criar um sistema distribuído para rastreio online de encomendas que permita ao utilizador o registo de conta e de encomendas bem como o acompanhamento da mesma ao longo de todo o seu trajeto. Também é pedido que exista a capacidade de operações de administração, sendo estas a de gestão das encomendas e de trajeto. Esta ferramenta terá de ter uma interface gráfica de modo a os utilizadores e os administradores poderem interagir com os serviços web SOAP de cada um.

2.1. Requisitos

- Registo de Utilizadores.
- Registo de Encomendas.
- Consulta do estado da encomenda por parte do utilizador.
- Visualização e Administração das encomendas via painel de admin.
- Interface gráfica web para interação com ambos os serviços web.
- O sistema deve utilizar serviços SOAP para comunicação.
- Base de Dados MySQL.
- Interface Gráfica baseada em Flask.
- Aplicação distribuída em containers, com Docker.

3. Análise e Desenho da Aplicação

3.1. Arquitetura Conceitual

A arquitetura da aplicação possui três camadas:

- **Camada de Apresentação**
  - GUI
- **Camada da "Business Logic"**
  - Web Service 1 (WS1) para os utilizadores.
  - Web Service 2 (WS2) para os administradores.
- **Camada dos Dados**
  - Base de Dados MySQL

3.1.1. Componentes

Frontend (GUI)

- Interface Web única para todos os utilizadores
- Controlo de acesso baseado nas funções (vistas cliente/administrador)
- Comunica com os serviços de back-end através de SOAP

Back-end

WS1 (Serviço do utilizador)

- Trata da autenticação e autorização dos utilizadores
- Gere o acompanhamento dos pacotes para os utilizadores regulares
- Operações principais:
  - Início de sessão/registo do utilizador
  - Listagem e pesquisa de pacotes
  - Acompanhamento do estado dos pacotes

WS2 (Serviço de administração)

- Operações administrativas
- Sistema de gestão de pacotes

- Operações principais:
    - Operações CRUD de pacotes
    - Gestão de utilizadores
    - Atualizações de rastreio a nível do sistema

Camada de base de dados

- Base de dados MySQL centralizada
- Três tabelas principais:
    - Utilizadores (autenticação/autorização)
    - Pacotes (itens de rastreio)
    - Informações de rastreio (actualizações de estado)

3.1.2 Modelo de Segurança

- Autenticação através de WS1
- Acesso baseado em funções:
    - **Clientes**: Limitado às operações do WS1
    - **Admins**: Acesso ao WS1 e ao WS2
- Hash de senha usando Argon2
- Gestão de sessão na camada GUI

3.1.3 Tecnologias utilizadas

Frontend: Flask + Bootstrap

Serviços de backend: Flask + Zeep + Spyne (SOAP)

Base de dados: MySQL 8.0

Plataforma de contentores: Docker + Docker Compose

Comunicação: SOAP/XML

4.2. Diagrama de Implantação

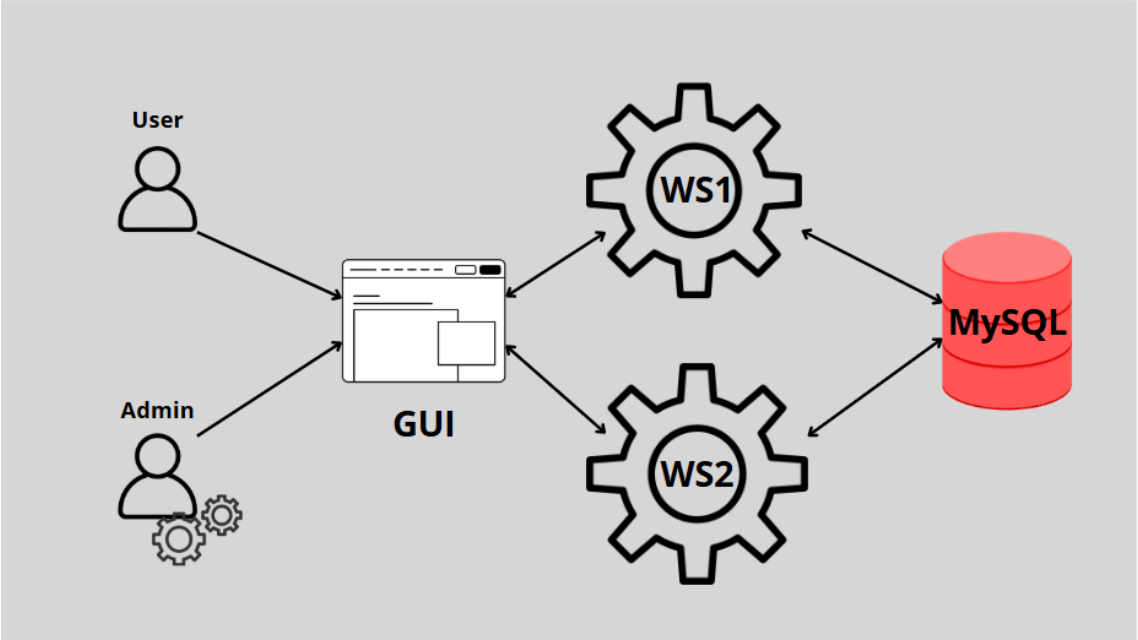


Figura 1 - Diagrama de Implementação

5. Implementação do Sistema Distribuído

O sistema distribuído foi dividido em quatro partes, como referido no capítulo anterior:

- GUI
- WS1
- WS2
- DB

Neste capítulo iremos abordar apenas os Web Services e a Base de Dados.

5.1 Tecnologias e Frameworks Utilizados

5.1.1 Flask

Apesar do framework Flask ser utilizado mais para o front-end da aplicação em si, também possui algumas vantagens no que toca ao back-end:

- Facilidade na definição de rotas e criação de endpoints RESTful, mesmo em aplicações que também consomem serviços SOAP.
- Integração com bibliotecas externas como o Spyne.

5.1.2 Spyne

Para a implementação dos serviços web SOAP WS1 e WS2, foi utilizado o framework Spyne, que oferece uma integração nativa com o Flask permitindo o uso de serviços SOAP dentro da aplicação. Algumas funcionalidades utilizadas foram:

- Definição de serviços via WSDL e validação de classes com Schemas XML;

```
spyne_app = Application([UserService],
    tns='sds.lab.user.v1',
    in_protocol=Soap11(validator='lxml'),
    out_protocol=Soap11()
)

spyne_wsgi_app = WsgiApplication(spyne_app)

flask_app.wsgi_app = DispatcherMiddleware(flask_app.wsgi_app, {
    '/ws1': spyne_wsgi_app
})
```

Figura 2 - Utilização do Spyne no ws1_user_service.py

- Definição de Tipos Complexos
    - Os tipos complexos no Spyne são estruturas de dados personalizadas que permitem definir objetos com múltiplos atributos para troca de dados via SOAP

```python
class PackageInfoAdmin(ComplexModel):
    _type_info = [
        ('id', Integer),
        ('name', Unicode),
        ('description', Unicode),
        ('sender_city', Unicode),
        ('destination_city', Unicode),
        ('is_tracked', Boolean),
        ('sender_username', Unicode),
        ('receiver_username', Unicode),
        ('creation_date', Unicode),
    ]
```

Figura 3 - Representação de uma encomenda com informações administrativas detalhadas

- Definição do serviço SOAP com ServiceBase
    - É uma classe base fornecida pelo Spyne para criar serviços SOAP. É o componente fundamental para definir serviços este tipo de serviços.

```python
class AdminService(ServiceBase):

    @rpc(_returns=Iterable(UserSelectionInfo))
    def getAllUsers(ctx):
        """Retorna lista de utilizadores para seleção."""
        users_data = db_get_all_users()
        return [UserSelectionInfo(**user) for user in users_data]
```

Figura 4 - Definição do serviço SOAP no WS2

Depois de ser definido o serviço, é usado o decorador *@rpc* para expor métodos como operações SOAP. Define também os tipos de entrada e saída de dados. Cada método recebe “ctx” como primeiro parâmetro de modo a permitir o acesso ao contexto da execução.

- Uso de tipos DateTime

```
@rpc(Integer, Unicode, DateTime, _returns=Boolean)
def registerPackageTracking(ctx, package_id, initial_city, initial_time):
    """Marca um pacote como rastreado e adiciona o ponto inicial."""
```

Figura 5 - Uso do DateTime do Spyne para rastreio da encomenda

- Gestão de erros com Fault

```
    Adiciona uma nova entrada de rastreamento a um pacote.
    if not all([package_id, city, time]):
        raise Fault(faultcode='Client', faultstring='Package ID, city, and time (DateTime) are required.')
    if package_id <= 0:
```

Figura 6 - Gestão de erros com Fault (Spyne)

5.13 MySQL Connector

A persistência dos dados foi garantida através de uma base de dados MySQL, cuja foi conectada aos serviços web através do MySQL connector. Os ficheiros db_util.py de ambos os serviços WS servem para tratar de tudo o que é interações com a base de dados.

```
DB_CONFIG = {
    'user': os.environ.get("MYSQL_USER"),
    'password': os.environ.get("MYSQL_PASSWORD"),
    'host': os.environ.get("MYSQL_HOST"),
    'database': os.environ.get("MYSQL_DATABASE"),
    'port': os.environ.get("MYSQL_PORT", 3306)
}

def get_db_connection():
    """Estabelece e retorna uma conexão MySQL."""
    try:
        conn = mysql.connector.connect(**DB_CONFIG)
        return conn
    except Error as e:
        print(f"Erro ao conectar ao MySQL: {e}")
        return None
```

Figura 7 - Configuração e conexão com a BD (db_utils.py)

Também nestes ficheiros é possível observar o mecanismo que trata do hashing das passwords com o algoritmo Argon 2.

```python
ph = PasswordHasher()

def hash_password(password):
    """Gera o hash de uma password usando Argon2."""
    return ph.hash(password.encode('utf-8'))

def check_password(hashed_password, plain_password):
    """Verifica se a password fornecida corresponde ao hash Argon2."""
    if not hashed_password or not plain_password:
        return False
    try:
        ph.verify(hashed_password, plain_password.encode('utf-8'))
        return True
    except VerifyMismatchError:
        return False
    except Exception as e:
        print(f"Erro ao verificar password com Argon2: {e}")
        return False
```

Figura 8 - Hashing de passwords com Argon2

Além disso também são aqui definidas as funções de acesso à BD que irão tratar das operações CRUD na BD.

```python
def db_user_login(username, password):
    conn = get_db_connection()
    if conn is None: return None
    cursor = conn.cursor(dictionary=True)
    user_info = None
    try:
        query = "SELECT id, password_hash, role FROM users WHERE username = %s"
        cursor.execute(query, (username,))
        user_record = cursor.fetchone()
        if user_record and check_password(user_record['password_hash'], password):
            user_info = {"user_id": user_record['id'], "role": user_record['role']}
    except Error as e: print(f"Erro na query db_user_login: {e}")
    except Exception as argon_e: print(f"Erro Argon2 em db_user_login: {argon_e}")
    finally:
        if cursor: cursor.close()
        if conn: conn.close()
    return user_info
```

Figura 9 - Função de acesso à BD para verificar credenciais

6. Implementação da Interface Gráfica (GUI)

Neste capítulo será abordada a Interface Gráfica (GUI), de que forma é que se integra no sistema distribuído e todos os aspetos que a rodeiam.

6.1. Tecnologias e Frameworks Utilizados

6.1.1 Flask

O Flask foi o framework escolhido pela sua simplicidade e capacidade de integração com os programas Spyne e Zeep. Além disso, o uso de templates Jinja2 permite o desenvolvimento rápido e eficaz da aplicação.

No ficheiro gui_app.py definimos as rotas Flask da seguinte forma:

- **Rotas Públicas**
  - /login
  - /register
  - /logout

- **Rotas de Cliente**
  - /dashboard/cliente
  - /package/<int:package_id>

- **Rotas de Administrador**
  - /dashboard/admin
  - /admin/package/add
  - /admin/package/delete/<int:package_id>
  - /admin/package/register_track/<int:package_id>
  - /admin/package/update_status/<int:package_id>

```python
@app.route('/')
def index():
    if 'user_id' in session:
        if session.get('role') == 'admin':
            return redirect(url_for('admin_dashboard'))
        else:
            return redirect(url_for('client_dashboard'))
    return redirect(url_for('login'))
```

Figura 10 - Rota Flask ('/')

```python
@app.errorhandler(404)
def page_not_found(e):
    return render_template('404.html'), 404

@app.errorhandler(500)
def internal_server_error(e):
    print(f"Internal Server Error: {e}")
    return render_template('500.html'), 500
```

Figura 11 - Gestão de Erros HTTP no GUI

6.1.2 Zeep

Se por um lado o Spyne serve para expor serviços SOAP em Python, por outro o **Zeep** é utilizado como cliente SOAP para consumir esses serviços. Além disso o Zeep permite facilitar a gestão dos dados XML, uma vez que este programa faz essa gestão de forma automática.

- Configuração do cliente SOAP

```python
client_ws1 = None
client_ws2 = None

try:
    client_ws1 = Client(WSDL_WS1, settings=settings, transport=transport)
    print("Cliente WS1 conectado.")
```

Figura 12 - Configuração do cliente SOAP para o WS1 na GUI

Por exemplo se quisermos chamar este serviço para o login:

```
    try:
        user_info = client_ws1.service.login(username=username, password=password)

        if user_info and user_info.user_id:
            session['user_id'] = user_info.user_id
            session['username'] = user_info.username
            session['role'] = user_info.role
            flash(f'Login bem sucedido como {user_info.role}!', 'success')
```

Figura 13 - Chamada do serviço client_ws1 na rota de login

6.3. Layout e Fluxo

6.3.1 Controlo de Acesso

```
def login_required(role="client"):
    """Verifica se o utilizador está logado e tem o role correto."""
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            if 'user_id' not in session:
                flash('Login necessário para aceder a esta página.', 'warning')
                return redirect(url_for('login', next=request.url))
            if session.get('role') != role and role != "any":
                flash(f'Acesso não autorizado. Role necessário: {role}.', 'danger')
                return redirect(url_for('index'))
            return f(*args, **kwargs)
        return decorated_function
    return decorator
```

Figura 14 - Controlo de acesso

6.3.2 Gestão de Sessões

```python
@app.route('/login', methods=['GET', 'POST'])
def login():
    if 'user_id' in session: return redirect(url_for('index'))

    if request.method == 'POST':
        username = request.form.get('username')
        password = request.form.get('password')

        if not client_ws1:
            flash('Erro crítico: Serviço de autenticação indisponível.', 'danger')
            return render_template('login.html')
        if not username or not password:
            flash('Utilizador e password são obrigatórios.', 'warning')
            return render_template('login.html')

        try:
            user_info = client_ws1.service.login(username=username, password=password)

            if user_info and user_info.user_id:
                session['user_id'] = user_info.user_id
                session['username'] = user_info.username
                session['role'] = user_info.role
                flash(f'Login bem sucedido como {user_info.role}!', 'success')

                next_page = request.args.get('next')
                return redirect(next_page or url_for('index'))
            else:
                flash('Resposta inesperada do serviço de login.', 'danger')
```

Figura 15 - Gestão da Sessão

6.4. Integração com o Sistema Distribuído

Como já referido anteriormente, o Zeep permite que os Web Services SOAP sejam recebidos pelo GUI e interpretados. A Figura 13 já faz referência a uma interação entre o GUI e o WS1. Mas também é possível apontar uma interação entre o GUI e o WS2:

```python
@app.route('/admin/package/add', methods=['GET', 'POST'])
@login_required(role="admin")
def add_package():
    users = []
    if client_ws2:
        try:
            users_data = client_ws2.service.getAllUsers()
            users = users_data if users_data else []
        except Exception as e:
            flash('Erro ao carregar lista de utilizadores.', 'warning')
            print(f"Erro getAllUsers: {e}")
```

Figura 16 - Interação entre o GUI e o WS2

7. Testes e Validação

7.1. Cenários de Teste

Para validar o funcionamento do sistema apresentado foram feitos os seguintes testes:

- Registo de um utilizador “Remetente”
- Registo de um utilizador “Destinatário”
- Registo de um utilizador “Cusco”
- Login com a conta pré-definida “Admin”
- Criar um pacote com Origem e Destino e adicionar tracking
- Visualizar o pacote no utilizador Destinatário
- Provar que um terceiro utilizador alheio não tem acesso ao pacote

7.2. Resultados Obtidos

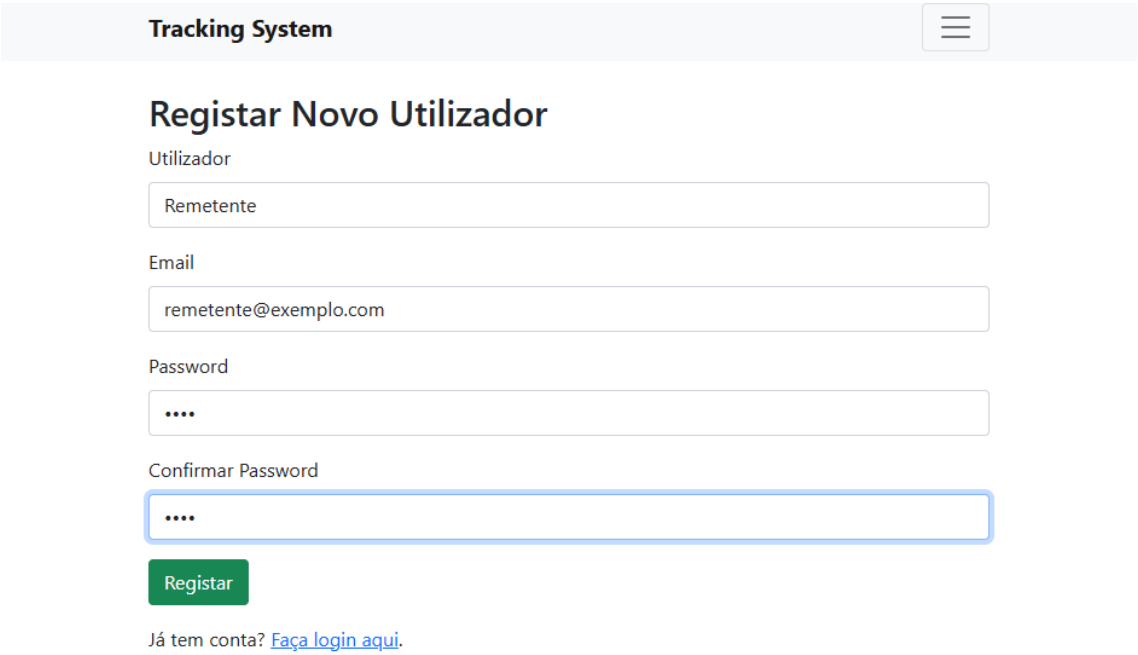


Figura 17 - Registo do Utilizador Remetente na App

Figura 18 - Utilizadores armazenados na BD MySQL

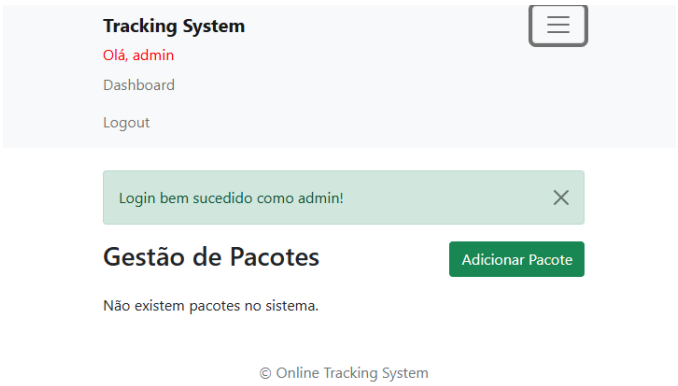


Figura 19 - Dashboard do Admin

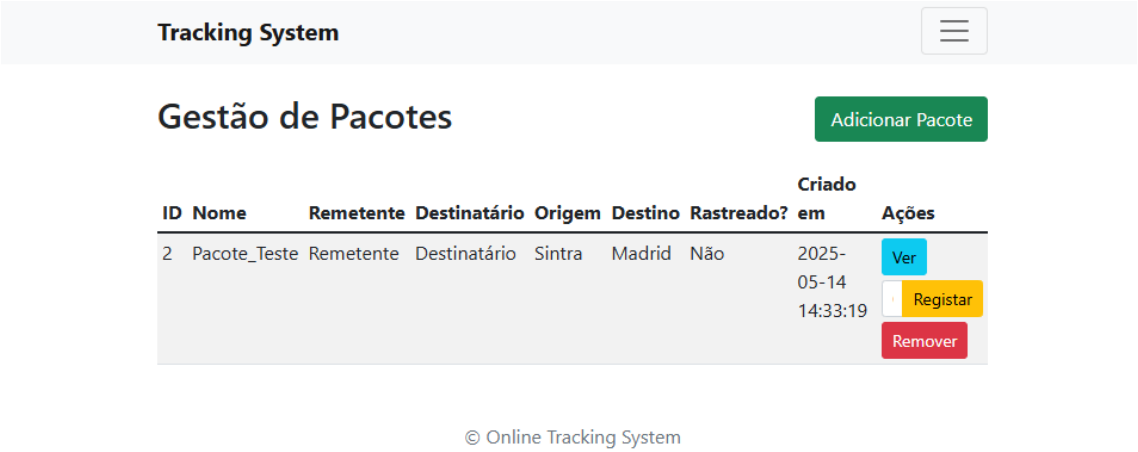


Figura 20 - Pacote Criado

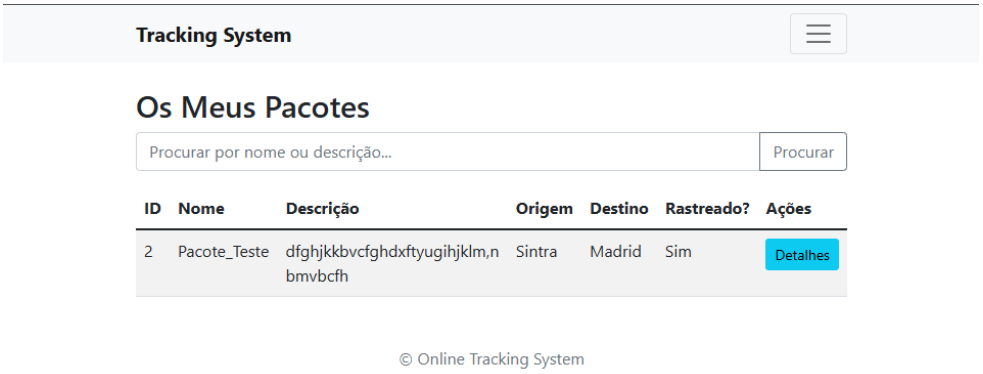


Figura 21 - Dashboard utilizador Destinatário

Figura 22 - Detalhes do Pacote

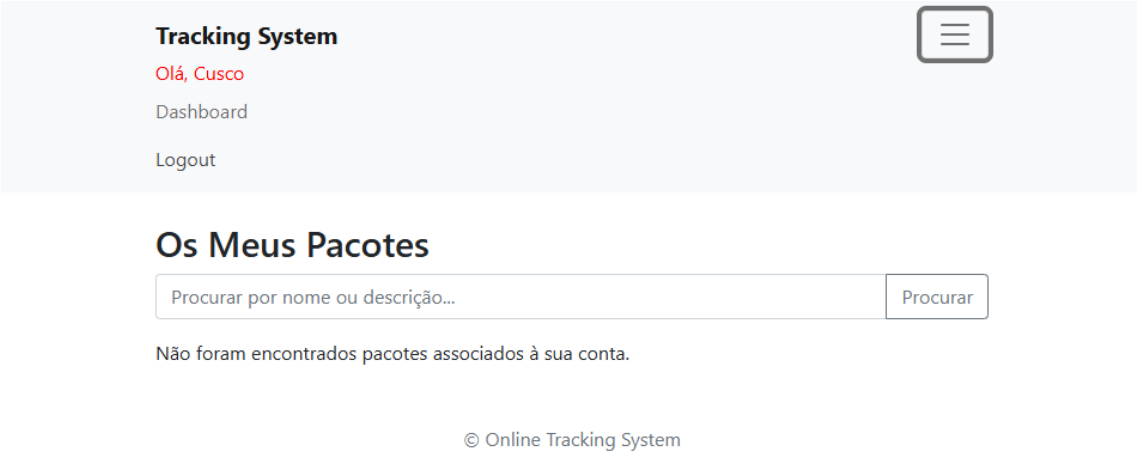


Figura 23 - Dashboard de um terceiro sem pacotes

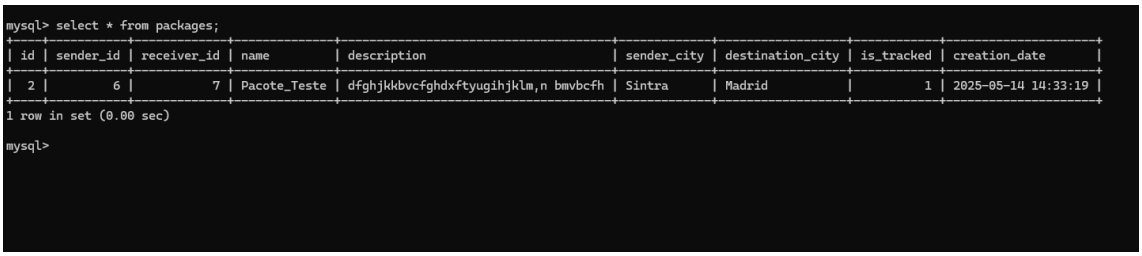


Figura 24 - Pacote na BD

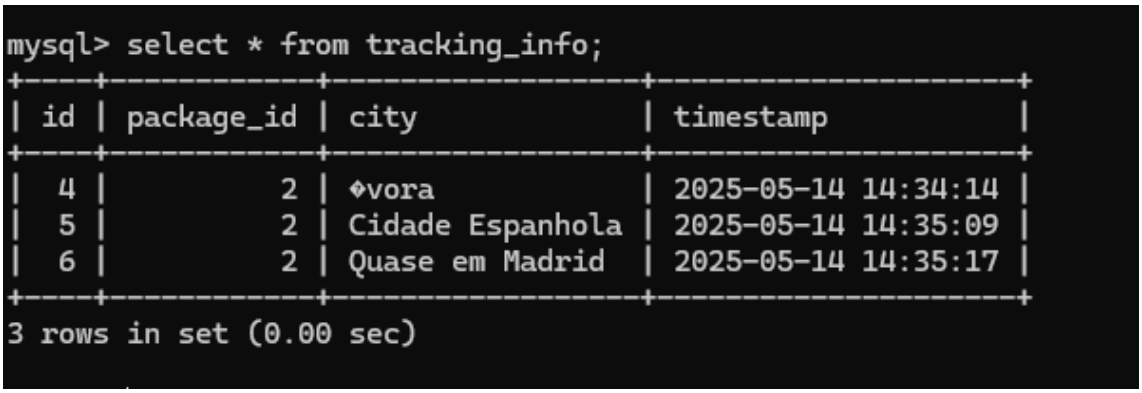


Figura 25 - Info do tracking do pacote criado

8. Conclusão

Este projeto demonstrou com sucesso a viabilidade e eficácia da construção de um sistema distribuído de rastreio de encomendas online, o "Online Tracking System", através da aplicação dos princípios da arquitetura orientada a serviços (SOA). A utilização de serviços web SOAP para operações dedicadas a utilizadores e administradores, juntamente com uma interface gráfica intuitiva, permitiu criar uma solução robusta e funcional.

A arquitetura implementada, com uma clara separação entre a camada de apresentação (GUI desenvolvida com Flask e Bootstrap), a camada de lógica de negócio (Web Services SOAP com Flask, Spyne e Zeep) e a camada de dados (Base de Dados MySQL), promoveu uma estrutura organizada e modular. Esta modularidade não só facilitou o desenvolvimento e a integração dos diferentes componentes, como também potencia a manutenibilidade e escalabilidade futuras do sistema. A containerização da aplicação com Docker simplificou a implantação e a gestão do ambiente distribuído.

Pontos Positivos do Projeto:

- **Implementação Efetiva de SOA:** A utilização de dois Web Services SOAP distintos (WS1 para utilizadores e WS2 para administradores) demonstrou uma correta aplicação dos conceitos de SOA, permitindo a criação de serviços coesos e independentes.
- **Comunicação Transparente e Normalizada:** A adoção de SOAP/XML para a comunicação entre a GUI e os web services assegurou uma interligação normalizada e fiável.
- **Interface Intuitiva e Funcional:** O desenvolvimento de uma interface gráfica única com Flask, capaz de se adaptar às funcionalidades de cliente e administrador, melhorou significativamente a usabilidade do sistema, permitindo interações eficazes para o registo de utilizadores, registo e consulta de encomendas, e gestão administrativa.
- **Segurança Considerada:** A implementação de hashing de passwords com Argon2 e a gestão de acesso baseada em funções (cliente/administrador) demonstram uma preocupação com a segurança dos dados e do sistema.

- **Persistência de Dados Fiável:** A utilização de uma base de dados MySQL centralizada, com interações geridas através do MySQL Connector, garantiu a correta armazenagem e recuperação da informação relativa a utilizadores, encomendas e rastreio.

Possíveis Melhorias Futuras:

- **Reforço dos Mecanismos de Segurança:** Poder-se-ia expandir a segurança com a implementação de autenticação de dois fatores (2FA), tokens de sessão mais robustos e encriptação dos dados em trânsito (HTTPS) e em repouso.
- **Expansão de Funcionalidades:** O sistema poderia ser enriquecido com notificações em tempo real sobre o estado das encomendas, integração com sistemas de mapas para visualização do trajeto, e funcionalidades analíticas para os administradores.
- **Otimização e Escalabilidade:** Avaliar e implementar mecanismos de caching, balanceamento de carga para os web services e otimizar as consultas à base de dados para suportar um maior volume de utilizadores e transações.
- **Adoção de Tecnologias Complementares:** Considerar a integração com filas de mensagens para operações assíncronas ou a exploração de APIs RESTful como alternativa ou complemento aos serviços SOAP para determinados casos de uso.

Em suma, o desenvolvimento do "Online Tracking System" cumpriu os objetivos propostos, demonstrando a aplicação prática e eficiente de sistemas distribuídos e arquiteturas orientadas a serviços na criação de uma solução web funcional e com potencial de evolução. O projeto proporcionou uma valiosa experiência na integração de diversas tecnologias como Flask, Spyne, Zeep, MySQL e Docker, resultando num sistema que responde eficazmente aos requisitos de rastreio de encomendas online, com benefícios claros na organização, funcionalidade e experiência do utilizador.