POLITÉCNICO DO PORTO
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Tópicos Avançados de Bases de Dados

Urban Transporation System

José Santos nº 1232153

TABDD
Tópicos Avançados de Bases de Dados

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Instituto Superior de Engenharia do Porto

Dezembro, 2025

Índice

1. Introdução

Este projeto visa a criação de uma aplicação dinâmica e inteligente para o sistema de autocarros urbanos de uma cidade. Capaz de monitorizar onde estão os veículos em tempo real, sugerir rotas mais rápidas, marcar linhas favoritas, guardar histórico de viagens, guardar paragens com a sua localização e nome, funções administrativas, alertas em caso de disrupção, capacidade de deixar feedback e outras funcionalidades.

Foi utilizada como inspiração 3 linhas de autocarros e 3 linhas de metro da cidade do Porto. No caso estão as linhas 500, 600 e 901 da STCP (cada linha de autocarro apenas tem 5 paragens e não todas as paragens) e as linhas A (azul), D (amarela) e E (violeta) do Metro do Porto (linhas de metro completas à data de fazer o seeding dos dados). Foi escolhida a cidade do Porto, visto ser a cidade onde este projeto foi criado e torna mais fácil a criação de features e a noção dos dados que estão a ser colocados sendo utilizador destes mesmos transportes.

2. Arquitetura e Tecnologias

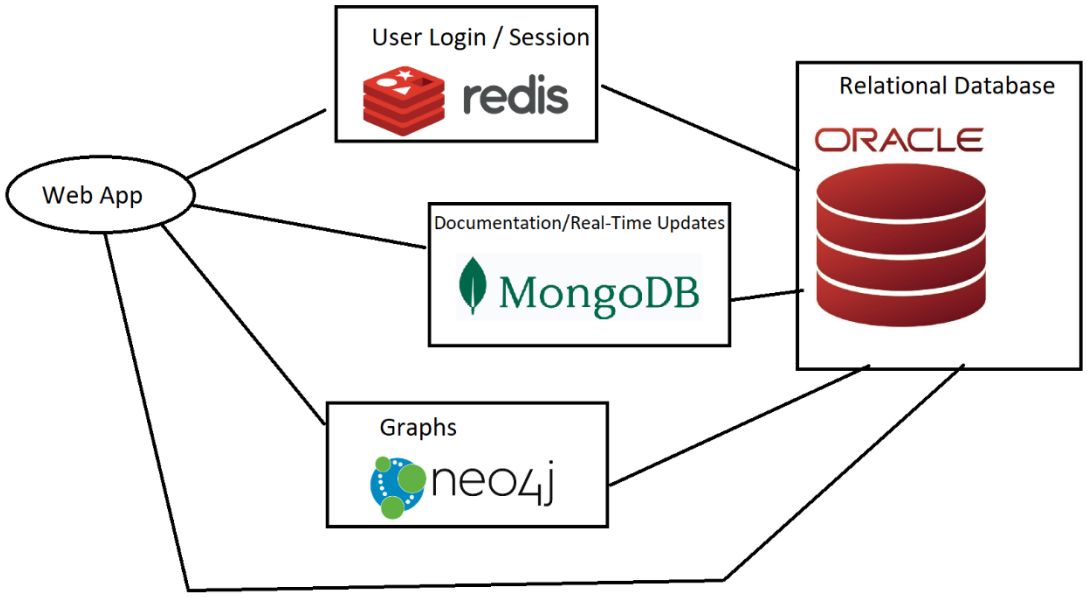


Figura 1- Arquitetura da WebAPP

Na primeira parte de entrega do projeto foi feita a referência que este projeto seria desenvolvido em Python com recurso ao FastAPI. A parte das bases de dados foi mantida num nível relacional com o Oracle, nível de login e manter sessão com o Redis, os grafos e sugestão de rotas atráves do Neo4J e toda a documentação mais flexível (como o perfil do utilizador e a marcação das suas linhas favoritas) e updates em tempo real pelo MongoDB, incluindo o live tracking dos veículos.

Para fazer a conexão com o Oracle, foi utilizado o servidor do DEI-ISEP. Já para as restantes bases de dados foi feito um setup local com recurso ao Docker para correr as bases de dados NoSQL. As imagens utilizadas são as seguintes:

```
PS C:\WINDOWS\system32> docker images

IMAGE          ID             DISK USAGE   CONTENT SIZE   EXTRA
mongo:latest   7245ffb851d1       1.24GB          324MB   U
neo4j:latest   9c1ccfeace87       1.07GB          402MB   U
redis:latest   43355efd2249        202MB           53MB   U
```

Figura 2 - Imagens Docker

Para inicializar estas imagens foram utilizados os seguintes comandos:

- docker run -d --name neo4j-local ` -p 7474:7474 -p 7687:7687 ` -e NEO4J_AUTH=neo4j/password `neo4j´
- docker run -d --name mongo-local -p 27017:27017 mongo
- docker run -d --name redis-local -p 6379:6379 redis

O servidor foi desenvolvido no editor Visual Studio Code e seguiu o formato indicado pelas aplicações FastAPI.

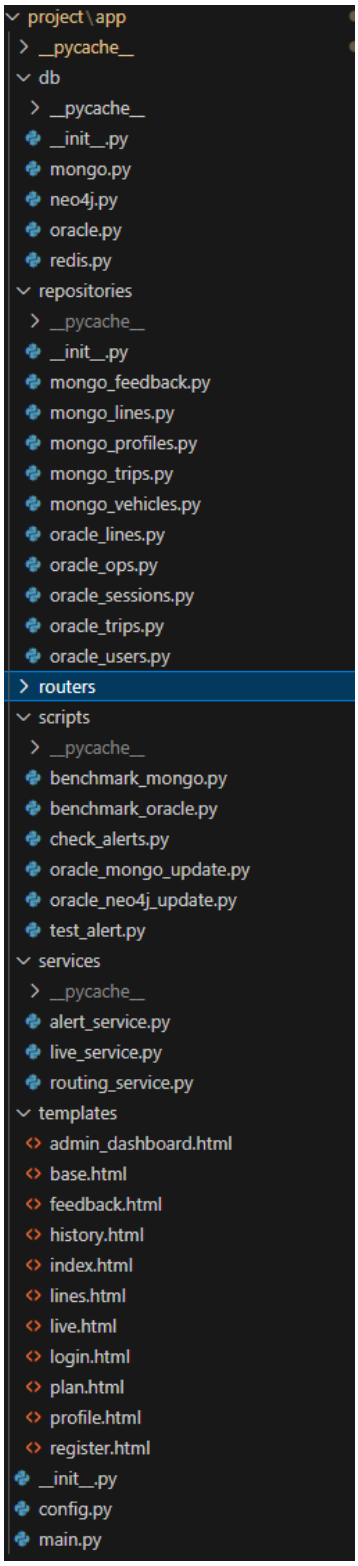


Figura 3 - Formatação aplicação

As configurações das quatro bases de dados estão guardadas no config.py, e as bases de dados são conectadas e correm nos ficheiros dentro da pasta “db” que comunica ao servidor como se conectar a elas e permitem a sua utilização para o consumidor utilizar a aplicação.

Dentro da pasta “repositories” estão funções para tornar a aplicação operacional através de queries às bases de dados e updates em tempo real.

Na pasta “scripts” estão scripts que apesar de não serem necessários para o uso normal da aplicação, podem ser necessários para debugging ou para fazer updates às bases de dados NoSQL, caso existam alterações à base de dados relacional.

Nos “services” encontram-se alguns dos features principais da aplicação, o alert_service que vai buscar as linhas favoritas do utilizador e envia alertas para o dashboard do utilizador caso existam disrupções. O live_service faz o seguimento em tempo-real dos veículos nas suas respetivas linhas. O routing_service cria as rotas para os utilizadores.

Na pasta dos “templates” estão todas as páginas html que o utilizador vai encontrar. Em algumas delas estão presentes scripts de JavaScript para facilitar o trabalho da parte do frontend.

Por fim, no main estão as funções principais das páginas e todas as APIs utilizadas, sendo que neste pedaço de código não existem queries diretas às bases de dados (apenas nos endpoints de teste à ligação para as bases de dados para efeitos de debugging).

3. Funcionalidades

Ao entrar no servidor existe uma página simples de Login com um link que redireciona para uma página simples de registo caso seja necessário criar um utilizador.

Figura 4 - Página de Login

Para o login / registo são utilizadas queries simples ao Oracle para verificar se o utilizador em que estão a tentar dar login existe. Caso exista, é criada uma sessão que é inserida. As sessões também são apagadas do Oracle quando terminam.

```python
pwd_context = CryptContext(schemes=["pbkdf2_sha256"], deprecated="auto")

def hash_password(password: str) -> str:
    return pwd_context.hash(password)

def verify_password(password: str, password_hash: str) -> bool:
    return pwd_context.verify(password, password_hash)

def get_user_by_email(email: str):
    sql = text("""
        SELECT user_id, email, password_hash, full_name, role, is_active
        FROM users
        WHERE email = :email
    """)
    with get_engine().begin() as conn:
        row = conn.execute(sql, {"email": email}).fetchone()
        return dict(row._mapping) if row else None

def get_user_by_id(user_id: str):
    sql = text("""
        SELECT user_id, email, password_hash, full_name, role, is_active
        FROM users
        WHERE user_id = :user_id
    """)
    with get_engine().begin() as conn:
        row = conn.execute(sql, {"user_id": user_id}).fetchone()
        return dict(row._mapping) if row else None

def create_user(email: str, password: str, full_name: str, role: str = "passenger"):
    user_id = str(uuid.uuid4())
    password_hash = hash_password(password)
    sql = text("""
        INSERT INTO users (
            user_id, email, password_hash, full_name, role, created_at, is_active
        )
        VALUES (
            :user_id, :email, :password_hash, :full_name, :role, SYSTIMESTAMP, 1
        )
    """)
```

Figura 5 - Query para login ou registo de user.

```python
# Update the timestamp to expire user session
def expire_user_session(session_id: str) -> int:
    sql = text("""
        UPDATE user_sessions
        SET expires_at = SYSTIMESTAMP
        WHERE session_id = :session_id
    """)
    with get_engine().begin() as conn:
        result = conn.execute(sql, {"session_id": session_id})
        return result.rowcount

# Delete the user session from DB
def delete_user_session(session_id: str) -> int:
    sql = text("""
        DELETE FROM user_sessions
        WHERE session_id = :session_id
    """)
    with get_engine().begin() as conn:
        result = conn.execute(sql, {"session_id": session_id})
        return result.rowcount

# retrieve active user session by session_id
def get_active_session(session_id: str) -> Optional[dict]:
    sql = text("""
        Create Jira Issue
        SELECT session_id, user_id, issued_at, expires_at
        FROM user_sessions
        WHERE session_id = :session_id
          AND expires_at > SYSTIMESTAMP
    """)
    with get_engine().begin() as conn:
        row = conn.execute(sql, {"session_id": session_id}).fetchone()
        return dict(row._mapping) if row else None
```

Figura 6 - Query para apagar sessão

Esta lógica é apoiada pelos APIs na main.py. O Redis está a ser também utilizado no main.

```python
def create_session(user_id: str, request: Request) -> str:

    token = str(uuid.uuid4())
    now = datetime.now(timezone.utc)
    expires_at = now + timedelta(seconds=settings.SESSION_TTL_SECONDS)

    user_agent = request.headers.get("user-agent")
    ip = request.client.host if request.client else None

    # Persists the session in Oracle DB
    oracle_sessions.create_user_session(
        session_id=token,
        user_id=user_id,
        Create Jira Issue
        issued_at=now,
        expires_at=expires_at,
        user_agent=user_agent,
        ip=ip,
    )

    # Persists the session in Redis cache
    redis_client.setex(
        SESSION_PREFIX + token,
        settings.SESSION_TTL_SECONDS,
        user_id,
    )

    return token
```

Figura 7 - Criação de sessão e utilização do Redis

Após o Login é iniciada a sessão que é mantida enquanto o utilizador estiver ativo. Caso contrário é desligada automaticamente passado 1 hora, deixando uma mensagem ao utilizador. A sessão é apagada do Redis e do Oracle.

Na página inicial do utilizador normal é possível navegar rapidamente para procurar as linhas, ver o histórico de viagens, adicionar favoritos ou deixar feedback. Também existe um header que permite a navegação para estas páginas, planear uma viagem, ver onde estão os transportes em tempo real e a possibilidade de deixar feedback. Também tem um botão para logout que apaga a sessão do Oracle e do Redis e por consequente termina a sessão do utilizador.

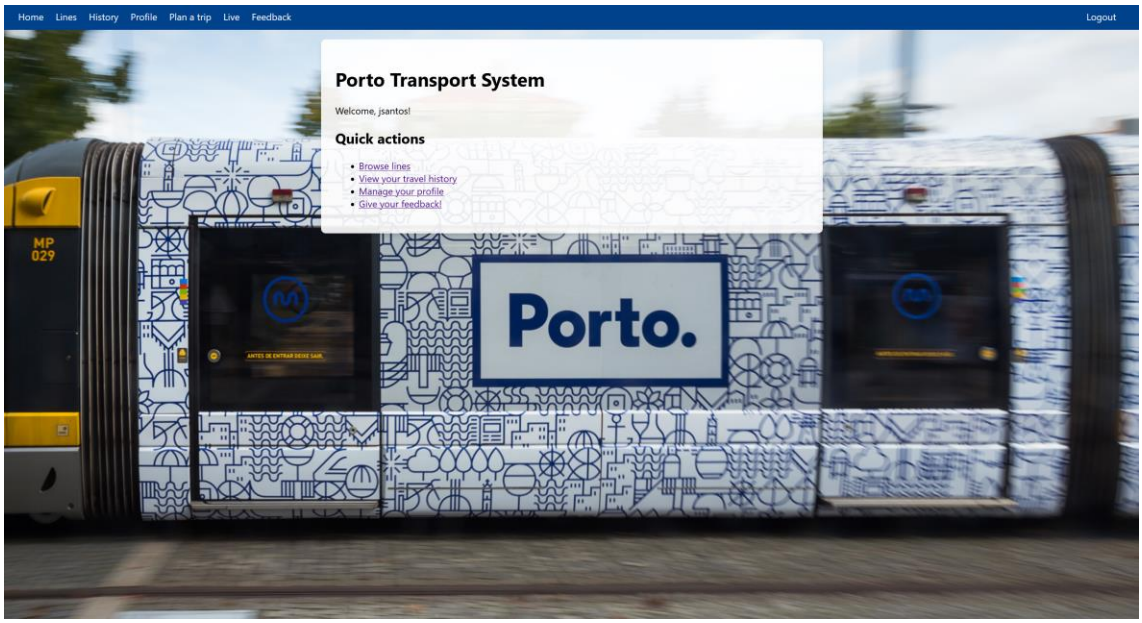


Figura 8 - Página inicial

Navegando para a página das linhas é possível ver as linhas suportadas pela aplicação e clicando nos detalhes de cada uma podemos ver a frequência em que os veículos passam e as suas paragens.

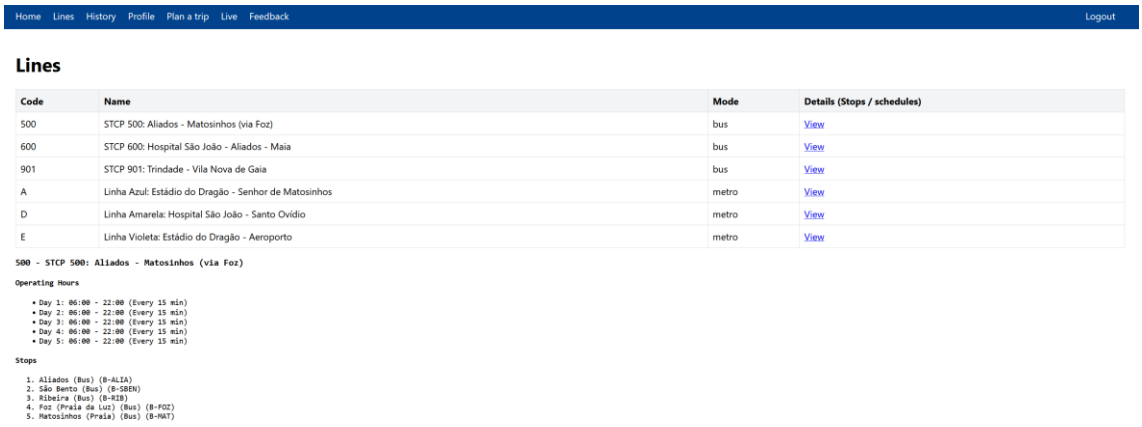


Figura 9 - Página das linhas

O botão “View” faz uma chamada ao API /oracle/lines/{line_id} que faz uma simples query ao Oracle para determinar as paragens associadas e a sua frequência em cada dia (os dias foram todos programados da mesma forma por razões de simplicidade)

```python
def get_line_details(line_id: str) -> Optional[Dict]:
    engine = get_engine()
    with engine.connect() as conn:
        line = conn.execute(
            text("SELECT line_id, code, name, line_mode, active FROM lines WHERE line_id = :lid"),
            {"lid": line_id}
        ).mappings().first()

        if not line:
            return None

        stops = conn.execute(
            text("""
                SELECT s.stop_id, s.code, s.name, st.scheduled_seconds_from_start
                FROM stop_times st
                JOIN stops s ON s.stop_id = st.stop_id
                WHERE st.line_id = :lid
                ORDER BY st.scheduled_seconds_from_start
            """),
            {"lid": line_id}
        ).mappings().all()

        schedules = conn.execute(
            text("""
                SELECT dow, TO_CHAR(start_time, 'HH24:MI') as start_str,
                       TO_CHAR(end_time, 'HH24:MI') as end_str, headway_minutes
                FROM line_schedules
                WHERE line_id = :lid
                ORDER BY dow
            """),
            {"lid": line_id}
        ).mappings().all()

    return {
        "line": dict(line),
        "stops": [dict(s) for s in stops],
        "schedules": [dict(s) for s in schedules]
    }
```

Figura 10 - Lógica e query ao Oracle quanto às linhas

Existe a possibilidade de planear uma viagem. Após o planeamento da viagem, o utilizador pode decidir se quer guardar a viagem no histórico ou não.

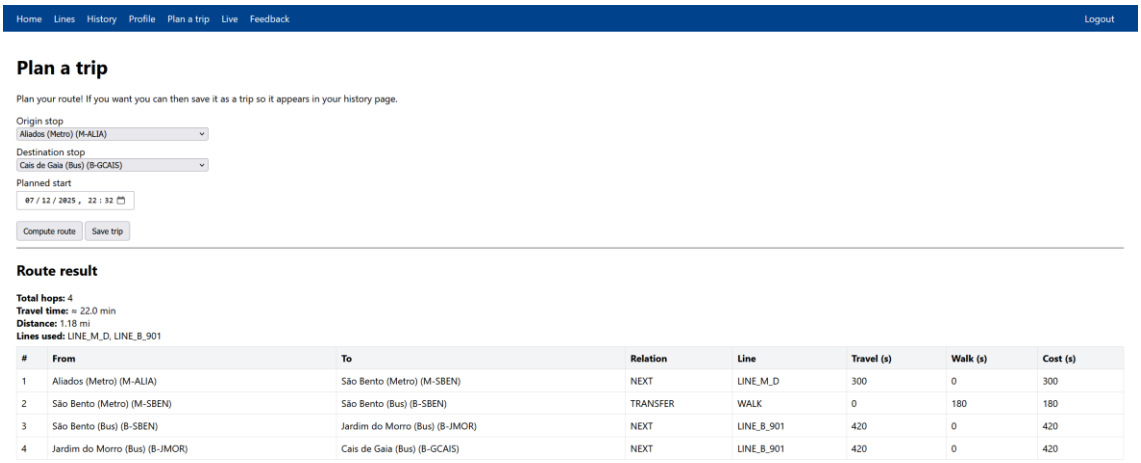


Figura 11 - Planeamento de viagem

Para fazer isto acontecer, é utilizada a API /route que faz um cálculo no Neo4J da menor distancia possível entre a paragem de origem selecionada e a paragem destino selecionada. Após o cálculo é retornado não só as paragens onde o utilizador irá passar, como também o tempo que irá demorar a fazer esta viagem, a distância e as linhas que vai utilizar. É desta forma que um dos polyglot features é utilizado com o routing_service e com o retorno dos detalhes das linhas (Oracle e MongoDB). Esta implementação também permite o utilizador fazer qualquer viagem entre estas paragens com baixa probabilidade de haver erros visto que a query feita ao Neo4J permite que o grafo seja lido de forma não direcionada.

```python
@app.get("/api/route")
def get_route(origin_stop_id: str, dest_stop_id: str, current_user=Depends(get_current_user)):
    user_id = current_user["user_id"]
    profile = mongo_profiles.get_or_create_profile(user_id)
    units = profile.get("prefs", {}).get("units", "metric")

    result = routing_service.find_best_route(origin_stop_id, dest_stop_id, units)

    if not result:
        raise HTTPException(status_code=404, detail="No route found")
    return result
```

Figura 12 - API Route

```python
def find_best_route(origin_id: str, dest_id: str, units: str = "metric"):
    # Neo4j Pathfinding
    driver = get_driver()
    with driver.session() as session:
        result = session.run("""
            MATCH (origin:Stop { stop_id: $oid }), (dest:Stop { stop_id: $did })
            MATCH p = allShortestPaths((origin)-[:NEXT|TRANSFER*..50]-(dest))
            WITH p, [r IN relationships(p) | CASE WHEN type(r)='TRANSFER' THEN 'TRANSFER' ELSE coalesce(r.line_id,'UNKNOWN') END] AS lids
            WITH p, reduce(sw=0, i IN range(1, size(lids)-1) | sw + CASE WHEN lids[i]<>lids[i-1] THEN 1 ELSE 0 END) AS switches
            ORDER BY switches ASC RETURN p AS path LIMIT 1
        """, oid=origin_id, did=dest_id).single()

    if not result:
        return None

    path = result["path"]
    nodes, rels = path.nodes, path.relationships

    segments, lines_used, stop_ids = [], set(), set()
    total_travel_s = 0

    # Build Segments
    for i, rel in enumerate(rels):
        from_n, to_n = nodes[i], nodes[i+1]
        line_id = rel.get("line_id")

        if line_id: lines_used.add(line_id)
        stop_ids.add(from_n.get("stop_id"))
        stop_ids.add(to_n.get("stop_id"))

        # Extract individual components
        avg_travel = rel.get("avg_travel_s") or 0
        walk_time = rel.get("walk_s") or 0
        cost = avg_travel + walk_time
        total_travel_s += cost

        segments.append({
            "rel_type": rel.type,
            "line_id": "WALK" if rel.type == "TRANSFER" else line_id,
            "from_stop": dict(from_n),
            "to_stop": dict(to_n),
            "avg_travel_s": avg_travel,
            "walk_s": walk_time,
            "cost_s": cost
        })
```

Figura 13 - Cálculo no Neo4J

Para guardar a viagem, é utilizada a API /trips que deixa a informação das linhas utilizadas, origem, destino, distancia e plano de início e fim de viagem.

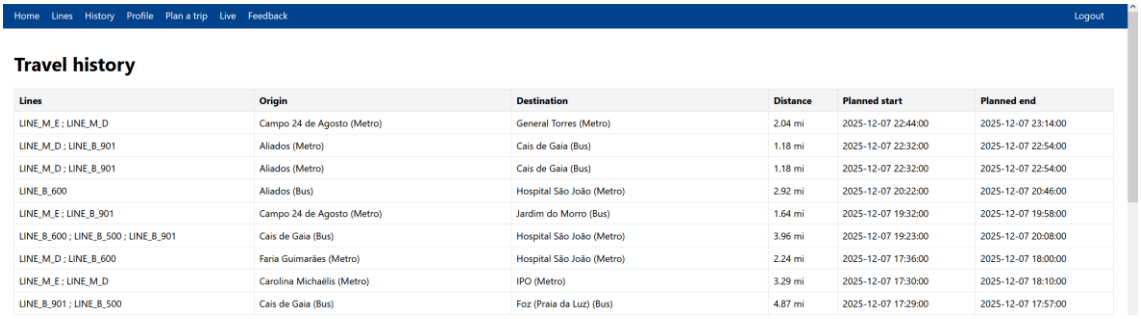


Figura 14 - Página History

```python
@app.post("/api/trips")
def create_trip(trip: TripCreate, current_user=Depends(get_current_user)):
    user_id = current_user["user_id"]
    now = datetime.now(timezone.utc)
    trip_id = str(uuid.uuid4())

    # Prepare Lines Used Logic
    if trip.lines_used and len(trip.lines_used) > 0:
        lines_used = trip.lines_used
    elif trip.line_id:
        lines_used = [trip.line_id]
    else:
        lines_used = []

    # Oracle write
    oracle_trips.create_trip(trip_id, user_id, trip)

    # MongoDB Write
    mongo_trip_doc = {
        "_id": trip_id,
        "user_id": user_id,
        "line_id": trip.line_id,
        "lines_used": lines_used,
        "origin_stop": trip.origin_stop_id,
        "dest_stop": trip.dest_stop_id,
        "planned_start": trip.planned_start,
        "planned_end": trip.planned_end,
        "created_at": now,
        "total_distance": trip.total_distance,
        "distance_unit": trip.distance_unit,
    }
    mongo_trips.create_trip(mongo_trip_doc)

    # Update User Profile
    recent_trip_summary = {
        "trip_id": trip_id,
        "at": now,
        "origin": trip.origin_stop_id,
        "dest": trip.dest_stop_id,
        "lines": lines_used,
        "dist": trip.total_distance,
        "unit": trip.distance_unit
    }
    mongo_trips.add_trip_to_user_history(user_id, recent_trip_summary)

    return {"ok": True, "trip_id": trip_id, "user_id": user_id}
```

Figura 15 - Lógica da API

Os dados são guardados no Oracle e nas coleções do MongoDB de forma detalhada.

```python
def create_trip(trip_id: str, user_id: str, data: Dict):
    sql = text("""
        INSERT INTO trips (
            trip_id, user_id, line_id, origin_stop_id, dest_stop_id,
            planned_start, planned_end, created_at
        ) VALUES (
            :tid, :uid, :lid, :oid, :did, :start, :end, :created
        )
    """)
    with get_engine().begin() as conn:
        conn.execute(sql, {
            "tid": trip_id, "uid": user_id, "lid": data.line_id,
            "oid": data.origin_stop_id, "did": data.dest_stop_id,
            "start": data.planned_start, "end": data.planned_end,
            "created": datetime.utcnow()
        })
```

Figura 16 - Create Trip Oracle

```python
# Insert a new trip document into MongoDB
def create_trip(data: Dict[str, Any]):
    trips_col = mongo_db["trips"]
    trips_col.insert_one(data)

# Updates the user's profile with the new trip.
def add_trip_to_user_history(user_id: str, trip_summary: Dict[str, Any]):
    profiles = mongo_db["user_profiles"]
    profiles.update_one(
        {"_id": user_id},
        {
            "$setOnInsert": {
                "favorites": {"lines": [], "stops": []},
                "prefs": {},
            },
            "$push": {
                "recentTrips": {
                    "$each": [trip_summary],
                    "$slice": -10,  # keep last 10
                }
            },
        },
        upsert=True,
    )
```

Figura 17 - Create Trip MongoDB

Também são utilizadas queries ao Mongo e ao Oracle para mostrar o histórico do utilizador.

```python
def get_user_history(user_id: str, limit: int = 20) -> List[Dict]:
    sql = text("""
        SELECT t.trip_id, t.planned_start, t.planned_end, t.line_id,
               o.name AS origin_name, d.name AS dest_name
        FROM trips t
        LEFT JOIN stops o ON t.origin_stop_id = o.stop_id
        LEFT JOIN stops d ON t.dest_stop_id = d.stop_id
        WHERE t.user_id = :uid
        ORDER BY t.planned_start DESC
        FETCH FIRST :lim ROWS ONLY
    """)
    with get_engine().connect() as conn:
        return [dict(row) for row in conn.execute(sql, {"uid": user_id, "lim": limit}).mappings().all()]
```

Figura 18 - Mostrar história do utilizador Oracle

```python
def get_trip_details_by_ids(trip_ids: List[str]) -> Dict[str, Dict]:

    if not trip_ids:
        return {}

    cursor = mongo_db["trips"].find(
        {"_id": {"$in": trip_ids}},
        {"_id": 1, "lines_used": 1, "total_distance": 1, "distance_unit": 1},
    )

    return {doc["_id"]: doc for doc in cursor}
```

Figura 19 - Mostrar histórico Mongo

Na página de perfil é possível ver todas as linhas e paragens disponíveis da aplicação. Em cada uma existe um botão para adicionar aos favoritos. Também é possível ajustar as preferências em termos de unidade de distância (nas páginas de planeamento de viagem e histórico é possível ver esta mudança) e se o utilizador quer ver notificações sobre disrupções nas linhas marcadas como favorito (notificações são visíveis na página principal apenas).

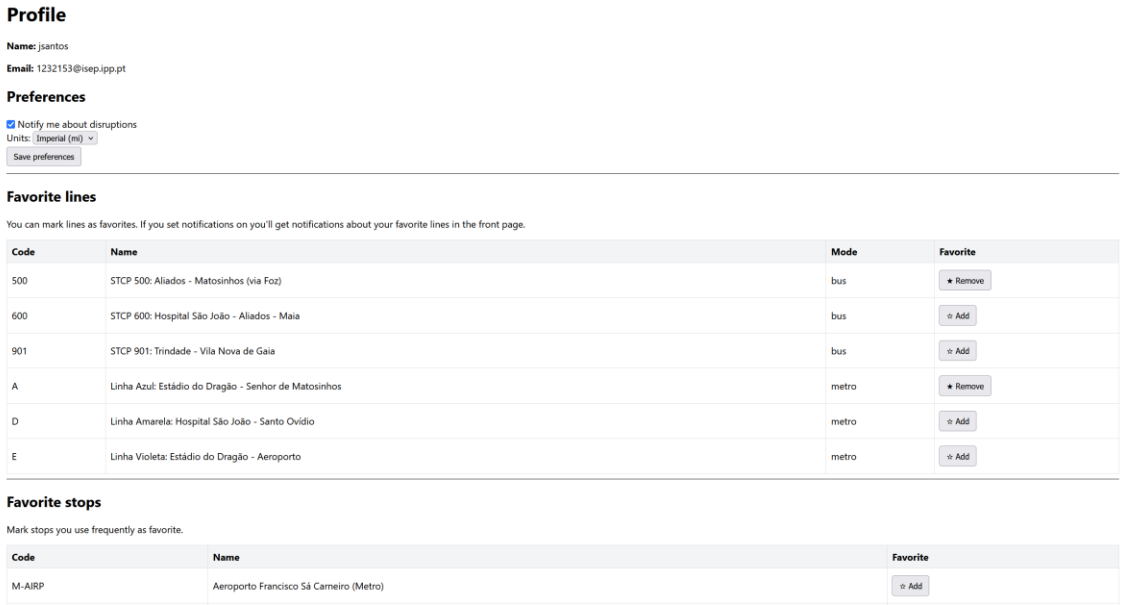


Figura 20 - Página de perfil de utilizador

Para as preferências do utilizador é usada a api /profile/prefs, para as linhas favoritas /profile/favorite/lines e para as paragens /profile/favorite/stops.

```python
# Let user set favorite lines
@app.post("/api/profile/favorites/lines")
def set_line_favorite(payload: LineFavoritePayload, current_user=Depends(get_current_user)):
    mongo_profiles.update_favorite_line(current_user["user_id"], payload.line_id, payload.favorite)
    return {"ok": True}

# favorite stops
@app.post("/api/profile/favorites/stops")
def set_stop_favorite(payload: StopFavoritePayload, current_user=Depends(get_current_user)):
    mongo_profiles.update_favorite_stop(current_user["user_id"], payload.stop_id, payload.favorite)
    return {"ok": True}

@app.post("/api/profile/prefs")
def set_prefs(payload: PrefsPayload,current_user=Depends(get_current_user),):
    mongo_profiles.update_preferences(current_user["user_id"], payload.notifyDisruptions, payload.units)
    return {"ok": True}
```

Figura 21 - Profile APIs

Estas APIs fazem queries simples às coleções do MongoDB.

```python
def update_favorite_line(user_id: str, line_id: str, is_favorite: bool):
    profiles = mongo_db.user_profiles
    operation = "$addToSet" if is_favorite else "$pull"

    # Ensure doc exists
    profiles.update_one(
        {"_id": user_id},
        {
            "$setOnInsert": {"prefs": {"notifyDisruptions": True, "units": "metric"}},
            operation: {"favorites.lines": line_id}
        },
        upsert=True
    )

def update_favorite_stop(user_id: str, stop_id: str, is_favorite: bool):
    profiles = mongo_db.user_profiles
    operation = "$addToSet" if is_favorite else "$pull"

    profiles.update_one(
        {"_id": user_id},
        {
            "$setOnInsert": {"prefs": {"notifyDisruptions": True, "units": "metric"}},
            operation: {"favorites.stops": stop_id}
        },
        upsert=True
    )

def update_preferences(user_id: str, notify: bool, units: str):
    profiles = mongo_db.user_profiles
    profiles.update_one(
        {"_id": user_id},
        {
            "$set": {
                "prefs.notifyDisruptions": notify,
                "prefs.units": units,
            },
            "$setOnInsert": {
                "favorites": {"lines": [], "stops": []},
                "recentTrips": [],
            },
        },
        upsert=True
    )
```

Figura 22 - Queries às APIs do perfil.

Quanto ao Feedback do utilizador é utilizada uma coleção no MongoDB apenas e uma página simples com a possibilidade de deixar um comentário e dar um rating à usabilidade da página e da satisfação do utilizador. Contudo, não existe um UI para ver este feedback e é apenas guardado na coleção.

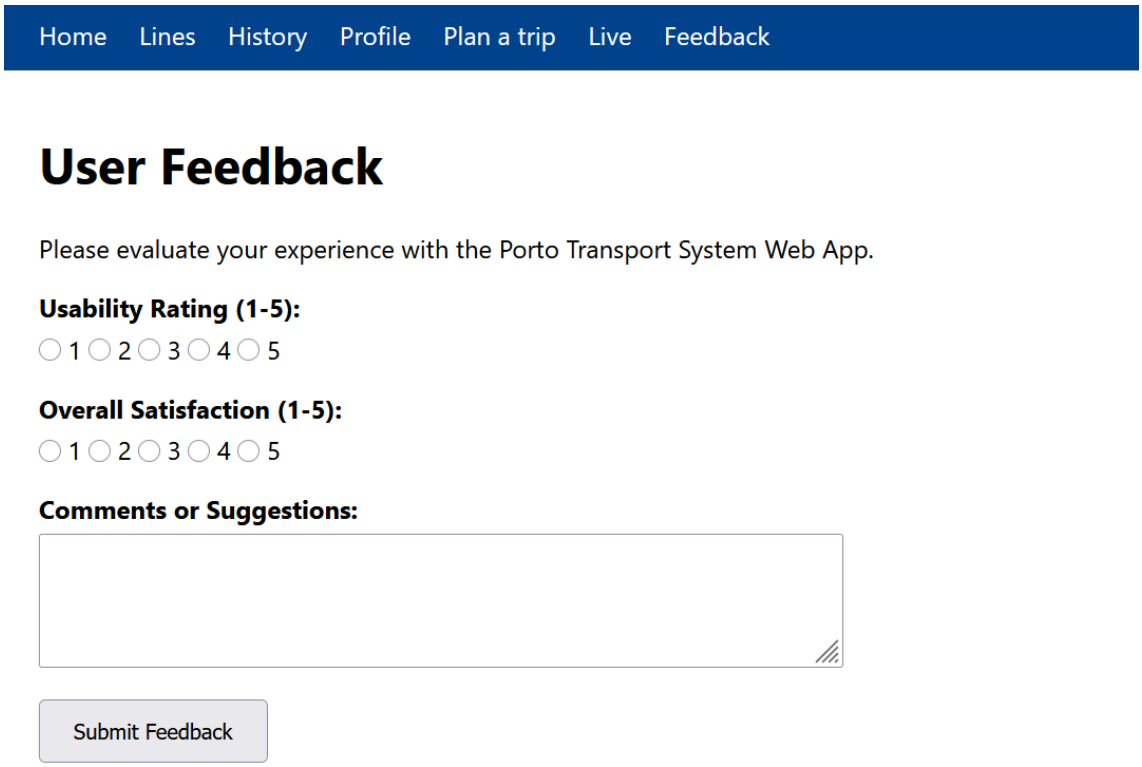


Figura 23 - Página de Feedback do User

Esta página faz um POST direto ao Mongo e cria uma entrada com o feedback.

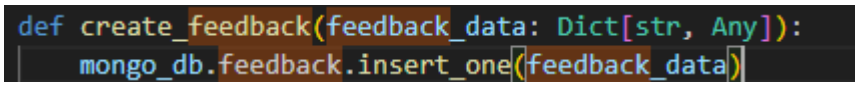
```python
@app.get("/feedback", response_class=HTMLResponse)
async def feedback_page(request: Request):
    return templates.TemplateResponse(
        "feedback.html",
        {"request": request},
    )

@app.post("/feedback", response_class=HTMLResponse)
def submit_feedback(
    request: Request,
    usability_rating: int = Form(...),
    satisfaction_rating: int = Form(...),
    comments: str = Form(""),
    current_user=Depends(get_current_user),
):
    feedback_data = {
        "user_id": current_user["user_id"],
        "usability_rating": usability_rating,
        "satisfaction_rating": satisfaction_rating,
        "comments": comments,
        "submitted_at": datetime.now(timezone.utc),
    }


    mongo_feedback.create_feedback(feedback_data)

    return templates.TemplateResponse(
        "feedback.html",
        {
            "request": request,
            "msg": "Thank you for your feedback!"
        },
    )
```

Figura 24 - Implementação FastAPI quanto ao feedback



```python
def create_feedback(feedback_data: Dict[str, Any]):
    mongo_db.feedback.insert_one(feedback_data)
```

Figura 25 - Inserção do Feedback

Também foi criada uma página para ver onde os veículos estão em tempo real. É possível definir de quanto em quanto tempo o utilizador quer ver a página a atualizar para ver a localização do veículo, a capacidade de lugares que o veículo tem, o seu modelo, de onde veio, qual a sua próxima paragem, quanto tempo até à sua próxima paragem e as suas coordenadas no momento a ver. O tracking dos veículos é feito atráves do MongoDB. Durante a realização do projeto verifiquei que teria sido melhor opção ter feito via Redis, visto que num projeto não académico não dará boa performance pela quantidade de queries

necessárias ao MongoDB para atualizar a posição do veiculo, algo que o Redis se daria melhor. Felizmente, como é um projeto académico com um sample pequeno de dados os impactos de performance não são notórios, contudo não escalariam bem para mais veículos, mais linhas e mais paragens. Sendo isto algo crucial a melhorar numa possível versão 2.

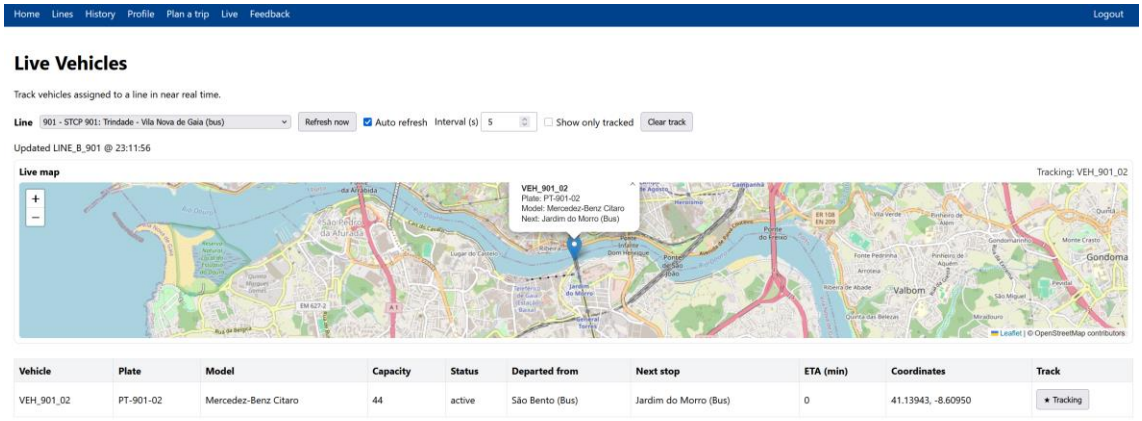


Figura 26 - Página dos veículos em tempo real.

Neste projeto académico, são utilizados dados fabricados, pelo que os dados ao vivo vêm de uma API internada /live/{line_id}, que por sua vez chama o serviço live_service, outro polyglot feature que utiliza o Oracle para ver quais são os veículos e condutores ativos e o MongoDB para todo o resto da implementação.

```python
@app.get("/api/live/{line_id}")
def live_line(line_id: str, _=Depends(get_current_user)):
    result = live_service.calculate_positions(line_id)

    if "error" in result:
        # If the service returned an error dict, handle it
        raise HTTPException(status_code=404, detail=result["error"])

    return result
```

Figura 27 - API /live

```python
def get_all_drivers() -> List[Dict]:
    sql = text("SELECT driver_id, license_no FROM drivers ORDER BY driver_id")
    with get_engine().connect() as conn:
        return [dict(row) for row in conn.execute(sql).mappings().all()]

def get_active_vehicles() -> List[Dict]:
    sql = text("SELECT vehicle_id, plate, model FROM vehicles WHERE active=1 ORDER BY plate")
    with get_engine().connect() as conn:
        return [dict(row) for row in conn.execute(sql).mappings().all()]

def create_assignment(assignment_id: str, driver_id: str, vehicle_id: str, line_id: str):
    sql = text("""
        INSERT INTO driver_assignments (
            assignment_id, driver_id, vehicle_id, line_id, start_ts
        ) VALUES (
            :aid, :did, :vid, :lid, SYSTIMESTAMP
        )
    """)
    with get_engine().begin() as conn:
        conn.execute(sql, {
            "aid": assignment_id, "did": driver_id, "vid": vehicle_id, "lid": line_id
        })

# active driver assignemnts by line
def get_active_assignments(line_id: str) -> Dict[str, Dict]:
    sql = text("""
        SELECT assignment_id, vehicle_id, driver_id
        FROM driver_assignments
        WHERE line_id = :lid
          AND start_ts <= SYSTIMESTAMP
          AND (end_ts IS NULL OR end_ts > SYSTIMESTAMP)
    """)

    with get_engine().connect() as conn:
        rows = conn.execute(sql, {"lid": line_id}).mappings().all()

    return {
        str(row["vehicle_id"]): {
            "assignment_id": str(row["assignment_id"]),
            "driver_id": str(row["driver_id"])
        }
        for row in rows
    }
```

Figura 28 – Queries ao Oracle

```python
def get_line_itinerary(line_id: str) -> List[Dict]:
    doc = mongo_db.lines.find_one(
        {"_id": line_id},
        {"itinerary": 1, "mode": 1, "code": 1, "name": 1}
    )
    if not doc:
        return []
    itinerary = doc.get("itinerary") or []
    # Ensure sorted by sequence
    return sorted(itinerary, key=lambda x: x.get("seq", 0))

# gets coordinates for stops
def get_stops_metadata(stop_ids: List[str]):
    cursor = mongo_db.stops.find(
        {"_id": {"$in": stop_ids}},
        {"_id": 1, "code": 1, "name": 1, "location": 1}
    )

    coords_map = {}
    meta_map = {}

    for d in cursor:
        sid = d["_id"]
        meta_map[sid] = {"stop_id": sid, "code": d.get("code"), "name": d.get("name")}

        loc = d.get("location") or {}
        coords = loc.get("coordinates")
        if coords and len(coords) == 2:
            coords_map[sid] = (float(coords[0]), float(coords[1]))

    return coords_map, meta_map

def get_vehicles_by_ids(vehicle_ids: List[str], line_id: str) -> List[Dict]:

    query = {"line": line_id}
    if vehicle_ids:
        query["_id"] = {"$in": vehicle_ids}
    return list(mongo_db.vehicles.find(query))

def update_vehicle_simulation(vehicle_id: str, sim_data: Dict, location: Optional[Dict], now_ts: datetime):
    update = {
        "$set": {
            "sim.idx": sim_data["idx"],
            "sim.segment_start_ts": sim_data["segment_start_ts"]
        }
    }
    if location:
        update["$set"]["lastKnown"] = {
            "ts": now_ts,
            "loc": location
        }

    mongo_db.vehicles.update_one({"_id": vehicle_id}, update)
```

Figura 29 – Queries ao MongoDB

```python
def calculate_positions(line_id: str):
    # Fetch Data
    itinerary = mongo_vehicles.get_line_itinerary(line_id)
    if not itinerary or len(itinerary) < 2:
        return {"error": "Itinerary missing or too short"}

    stop_ids = [i["stop_id"] for i in itinerary]
    coords_map, meta_map = mongo_vehicles.get_stops_metadata(stop_ids)

    assignment_map = oracle_ops.get_active_assignments(line_id)
    active_vehicle_ids = list(assignment_map.keys())

    vehicles = mongo_vehicles.get_vehicles_by_ids(active_vehicle_ids, line_id)

    now = datetime.now(timezone.utc)
    enriched_vehicles = []

    if not active_vehicle_ids:
        note = "No active driver assignments now"
    else:
        note = None

    # Simulate vehicles
    for v in vehicles:
        sim = v.get("sim") or {}
        idx = int(sim.get("idx") or 0)
        seg_start = sim.get("segment_start_ts") or now

        # Calculate new position
        new_idx, new_seg_start, elapsed, travel_s = _advance_segment(itinerary, idx, seg_start, now)

        from_stop = itinerary[new_idx]["stop_id"]
        to_stop = itinerary[new_idx+1]["stop_id"]

        progress = 0.0
        loc_obj = None

        if from_stop in coords_map and to_stop in coords_map and travel_s > 0:
            progress = max(0.0, min(1.0, elapsed / travel_s))
            lon, lat = _interpolate(coords_map[from_stop], coords_map[to_stop], progress)
            loc_obj = {"type": "Point", "coordinates": [lon, lat]}

        # Update vehicle simulation state
        mongo_vehicles.update_vehicle_simulation(
            v["_id"],
            {"idx": new_idx, "segment_start_ts": new_seg_start},
            loc_obj,
            now
        )
```

Figura 30 - Cálculo das posições em "tempo real"

```
    remaining_s = max(int(travel_s - elapsed), 0)
    assign_info = assignment_map.get(str(v.get("_id")))

    payload = {
        "vehicle_id": v.get("_id"),
        "plate": v.get("plate"),
        "model": v.get("model"),
        "capacity": v.get("capacity"),
        "line_id": line_id,
        "lastKnown": {"ts": now.isoformat(), "loc": loc_obj} if loc_obj else None,
        "departed_stop": meta_map.get(from_stop),
        "next_stop": meta_map.get(to_stop),
        "eta_to_next_stop_s": remaining_s,
        "status": "active" if assign_info else "inactive"
    }

    if assign_info:
        payload.update(assign_info)

    enriched_vehicles.append(payload)

return {
    "line_id": line_id,
    "ts": now.isoformat(),
    "vehicles": enriched_vehicles,
    "note": note
}
```

Figura 31 - Cálculo das posições em "tempo real" (2)

Foi criada uma página de administrador que só utilizadores com role de administrador podem aceder. Esta página permite criar alertas para certas linhas, mudar os condutores de veículos e linha. Também é possível "banir" utilizadores de utilizarem a Web App sendo-lhes impossível de dar login sem o admin voltar a dar permissão para tal.

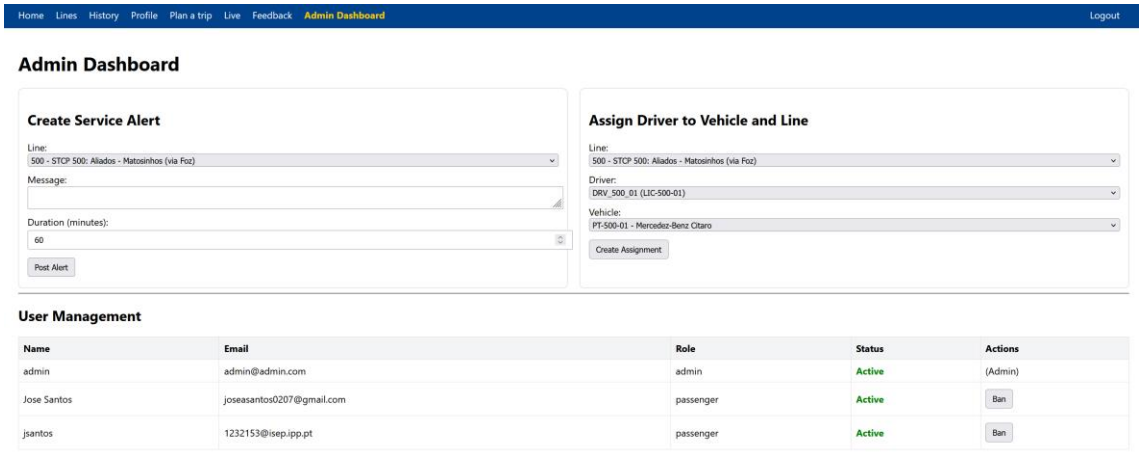


Figura 32 - Dashboard de administrador

Existem vários endpoints na API /admin para fazer esta queries.

```python
# ####################
# Admin API Endpoints
# ####################

@app.post("/api/admin/alerts")
def create_alert(payload: AdminAlertCreate, _=Depends(get_current_admin)):
    now = datetime.now(timezone.utc)
    end_time = now + timedelta(minutes=payload.duration_minutes)

    new_alert = {
        "msg": payload.msg,
        "from": now,
        "to": end_time
    }

    modified_count = mongo_lines.add_alert(payload.line_id, new_alert)

    return {"ok": True, "modified": modified_count}

@app.post("/api/admin/users/{user_id}/status")
def toggle_user_status(user_id: str, payload: UserStatusUpdate, _=Depends(get_current_admin)):
    oracle_users.update_user_status(user_id, payload.is_active)
    return {"ok": True}

@app.post("/api/admin/assignments")
def create_assignment(payload: DriverAssignmentCreate, _=Depends(get_current_admin)):
    assignment_id = "ASG_" + str(uuid.uuid4())[:8]
    oracle_ops.create_assignment(assignment_id, payload.driver_id, payload.vehicle_id, payload.line_id)
    return {"ok": True, "assignment_id": assignment_id}
```

Figura 33 - Admin API

```python
def add_alert(line_id: str, alert: Dict[str, Any]) -> int:

    result = mongo_db.lines.update_one(
        {"_id": line_id},
        {"$push": {"alerts": alert}}
    )

    # Fallback, tries updating by field 'line_id' if _id didn't match
    if result.matched_count == 0:
        result = mongo_db.lines.update_one(
            {"line_id": line_id},
            {"$push": {"alerts": alert}}
        )

    return result.modified_count
```

Figura 34 - Adicionar alertas

```python
def update_user_status(user_id: str, is_active: bool):
    sql = text("""
        UPDATE users
        SET is_active = :is_active
        WHERE user_id = :user_id
    """)
    with get_engine().begin() as conn:
        conn.execute(sql, {"is_active": 1 if is_active else 0, "user_id": user_id})
```

Figura 35 - Atualizar o estado de um utilizador

```python
def create_assignment(assignment_id: str, driver_id: str, vehicle_id: str, line_id: str):
    sql = text("""
        INSERT INTO driver_assignments (
            assignment_id, driver_id, vehicle_id, line_id, start_ts
        ) VALUES (
            :aid, :did, :vid, :lid, SYSTIMESTAMP
        )
    """)
    with get_engine().begin() as conn:
        conn.execute(sql, {
            "aid": assignment_id, "did": driver_id, "vid": vehicle_id, "lid": line_id
        })
```

Figura 36 - Alterar os condutores

Para demonstrar o sistema de alertas, podemos ver que se o administrador criar um alerta para uma linha em que um utilizador tenha marcado como favorito e tenha as notificações ativas, irá aparecer na página principal quando navegar até lá.

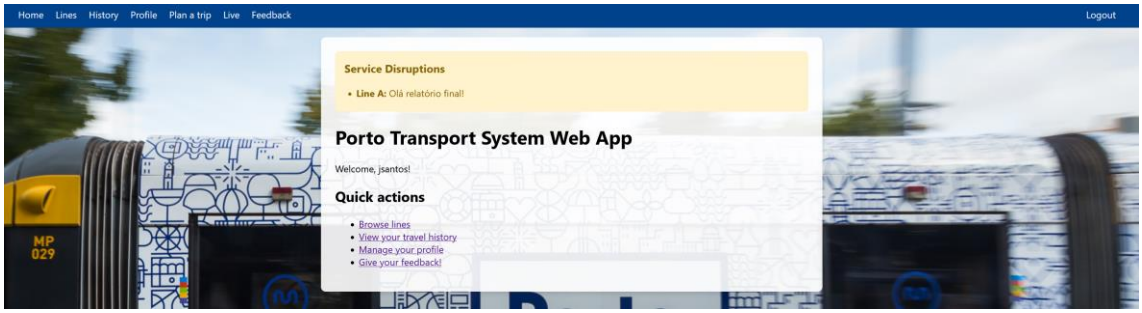


Figura 37 - Sistema de alertas

4. Utilização de Índices

Tanto a minha implementação no Oracle como no Mongo já incluíam índices à priori. Por tal, criei dois scripts para demonstrar esta parte que se encontram no projeto em scripts/benchmark_*.py

Para o Oracle temos este script com as seguintes queries:

```
sql_with_index = text("""
    SELECT count(*)
    FROM driver_assignments
    WHERE line_id = :lid
      AND start_ts > TO_TIMESTAMP(:ts, 'YYYY-MM-DD HH24:MI:SS')
""")

# forces Oracle to ignore the index and do a full table scan
sql_no_index = text("""
    SELECT /*+ NO_INDEX(driver_assignments idx_driver_assignments_active) */ count(*)
    FROM driver_assignments
    WHERE line_id = :lid
      AND start_ts > TO_TIMESTAMP(:ts, 'YYYY-MM-DD HH24:MI:SS')
""")

params = {"lid": "LINE_M_A", "ts": "2024-01-01 00:00:00"}
iterations = 2000
```

Figura 38 - Benchmark do Oracle

Esta query faz 2000 iterações visto que como a base de dados é relativamente pequena as diferenças de performance com ou sem índice não serão significativas.

```
PS C:\Users\josea\Desktop\Mestrado\TABDD\project> C:/Python312/python.exe -m app.scripts.benchmark_oracle
--- ORACLE BENCHMARK (2000 iterations) ---
WITH Index:    28.5879 seconds
WITHOUT Index: 28.7705 seconds
Improvement:   0.6% faster with index
```

Figura 39 - Resultados benchmark Oracle

Vimos aqui que com o índice a query foi 0.6% mais rápido, o que significa que se aumentássemos a quantidade de queries e de dados o índice pouparia muito tempo e claramente ajudam na performance, não obrigando à base de dados a fazer um full table scan ver todas as linhas e apenas fazendo um range scan.

Para o MongoDB temos as seguintes queries:

```python
def run_benchmark():
    col = mongo_db["vehicles"]
    line_query = "LINE_M_A"
    iterations = 2000

    print(f"--- MONGODB BENCHMARK ({iterations} iterations) ---")

    # Ensure Index Exists
    col.create_index("line")

    # Measure WITH Index first
    start = time.time()
    for _ in range(iterations):
        _ = list(col.find({"line": line_query}))
    end = time.time()
    time_index = end - start
    print(f"WITH Index:    {time_index:.4f} seconds")

    # Drop Index to simulate "Before Optimization"
    try:
        col.drop_index("line_1") # Default mongo name for {line: 1} is line_1
    except Exception as e:
        print(f"Warning: Could not drop index (might not exist): {e}")

    # Measure WITHOUT Index
    start = time.time()
    for _ in range(iterations):
        _ = list(col.find({"line": line_query}))
    end = time.time()
    time_no_index = end - start
    print(f"WITHOUT Index: {time_no_index:.4f} seconds")

    # Restore Index
    col.create_index("line")
    print("Index restored.")
```

Figura 40 - Benchmark Mongo

```
PS C:\Users\josea\Desktop\Mestrado\TABDD\project> C:/Python312/python.exe -m app.scripts.benchmark_mongo
--- MONGODB BENCHMARK (2000 iterations) ---
WITH Index:    1.2925 seconds
WITHOUT Index: 1.4120 seconds
Index restored.
Improvement:   8.5% faster with index
```

Figura 41 - Resultados Benchmark MongoDB

A criação do índice melhora o query em 8.5%, não obrigando à base de dados a ver todas as linhas e apenas indo buscar a necessária para a query. Mesmo sendo uma base de dados pequena é possível ver uma grande melhoria de performance, algo que seria certamente muito notório num caso real.

5. Conclusões e Melhorias

Este trabalho demonstra uma implementação de Polyglot Architecture e demonstra as melhores partes de cada base de dados para este trabalho. A velocidade em criar sessões do Redis, a flexibilidade do MongoDB, a capacidade de cálculo em grafo do Neo4J e a manutenção de integridade de dados do Oracle.

É também visível que os índices melhoram significativamente em performance e sem eles não é possível manter rapidez ao escalar a quantidade de dados e de utilizadores no serviço, tendo conseguido 0.6% de melhoria no Oracle e 8.5% no MongoDB em testes de baixa-escala.

Neste trabalho era possível ter melhorado no live service, tendo utilizado o Redis em vez do MongoDB para fazer o cálculo em tempo real dos veículos, algo que iria deixar o serviço mais rápido. A utilização de tecnologias mais modernas para o front-end também teria melhorado e possibilidade para os administradores verem o feedback que os utilizadores possam dar.