ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

# Trabalho Prático Final – Fase 2


48089 André Filipe Pina Páscoa

48280 André Filipe do Pilar de Jesus

48287 Nyckollas Brandão

Orientador Engenheiro Walter Vieira


Relatório de final realizado no âmbito de Sistemas de Informação,
do curso de licenciatura em Engenharia Informática e de Computadores
Semestre de Verão 2021/2022

Junho de 2022

# Resumo

O presente trabalho tem como principal objetivo o desenvolvimento de uma camada de acesso a dados, utilizada numa aplicação na linguagem Java. Durante a implementação do trabalho, utilizou-se a ferramenta JPA (Java Persistence API) e um subconjunto dos padrões de desenho, como DataMapper, Repository e UnitOfWork, estudados em aula.

Numa primeira fase, implementou-se o modelo de dados, previamente desenhado na primeira fase do projeto, em Java, com recurso a JPA.

Numa segunda fase, desenvolveu-se a camada de acesso a dados, utilizada no modelo. Com recurso à camada de acesso a dados desenvolvida, implementaram-se as funcionalidades, descritas na fase 1 do projeto, em Java.

Numa terceira fase, reimplementou-se uma das funcionalidades usando optimistic locking, realizando testes unitários para testar o funcionamento da mesma.

Apresenta-se também no documento, a discussão das alternativas de modelação e as razões de escolha das soluções apresentadas.

Após a realização do trabalho, concluiu-se que os objetivos de aprendizagem foram alcançados, produzindo os resultados pretendidos e conhecimentos da matéria em estudo foram adquiridos com sucesso.

**Palavras-chave:** camada de acesso a dados, processamento transacional, padrão de acesso a dados, ciclo de vida das entidades, optimistic locking

# Abstract

This work has the primary goal the development of a data access layer, used in a Java language application. During the implementation of the work, we used the JPA (Java Persistence API) tool and a subset of the design patterns, such as DataMapper, Repository and UnitOfWork, studied in class.

In a first phase, the data model, previously designed in the first phase of the project, was implemented in Java, using JPA.

In a second phase, the data access layer, used in the data model, was developed. Using the developed data access layer, the functionalities described in phase 1 of the project were implemented in Java.

In a third phase, one of the functionalities was re-implemented using optimistic locking, performing unit tests to test its functioning.

The current document presents a discussion on the model alternatives and the reasons for the choice of the presented solutions.

After the end of this work, it's concluded that the learning goals were reached, producing the expected results and knowledge on the studied subject was acquired successfully.

**Keywords:** data access layer, transactional processing, data access pattern, entity lifecycle, optimistic locking

# Índice

# 1. Introdução

Este trabalho teve como principal objetivo o estudo e desenvolvimento de uma camada de acesso a dados, aplicada ao modelo de dados desenvolvido na fase anterior. Esta camada é utilizada numa aplicação na linguagem Java, com recurso à tecnologia JPA (Java Persistence API).

Inicialmente, implementou-se o modelo de dados, previamente desenhado na primeira fase do projeto, em Java, com recurso a JPA. Para cada entidade do modelo, foi implementada uma classe Java que representa essa mesma entidade. Cada classe tem como propriedades os atributos pertencentes à entidade, e alguns métodos de encapsulamento dos mesmos.

Com o modelo de dados implementado em Java, começou-se o desenvolvimento da camada de acesso a dados. Nesta camada, são implementados padrões de acesso a dados lecionados nas aulas, entre eles Mapper, Repository e PersistenceManager. Estes padrões são responsáveis por realizar operações sobre as respetivas entidades, como por exemplo operações CRUD (Create, Read, Update, Delete). Com a camada de acesso a dados implementada, desenvolveram-se as funcionalidades descritas na fase 1 do projeto em Java. Estas funcionalidades foram implementadas em procedimentos armazenados, funções, gatilhos e vistas.

Posteriormente, reimplementou-se uma das funcionalidades utilizando a técnica optimistic locking, que é uma técnica usada em aplicações com bases de dados SQL, que não mantém locks de linha durante leituras, updates ou deleções.

Numa fase final, as peças de software foram devidamente testadas, comprovando o seu correto funcionamento.

Apresenta-se também no documento, a discussão das alternativas de modelação e as razões de escolha das soluções apresentadas.

Após a realização do trabalho, concluiu-se que os objetivos de aprendizagem foram alcançados, produzindo os resultados pretendidos e conhecimentos da matéria em estudo foram adquiridos com sucesso.

## 1.1 Estrutura do projeto

Neste subcapítulo está descrita a estrutura de diretorias do projeto desenvolvido.
A diretoria do projeto está organizada em duas subdiretorias:

- "data-model" - contém os modelos de dados conceptual e relacional;
- "src" - contém todo o código Java desenvolvido.

Na diretoria "src" existem três subdiretorias:

- "main/java/pt/isel" - contém o código Java desenvolvido na fase 2 do projeto;
- "test/java/pt/isel" - contém os testes unitários do respetivo código Java;
- "main/sql" - contém todo o código PL/pgSQL desenvolvido.

A diretoria "main/java/pt/isel" tem as seguintes subdiretorias:

- "dal" – (data access layer) camada de acesso a dados;
- "dataProcessors" – processadores de dados GPS;
- "model" – modelo de dados implementado com recurso a JPA;
- "presentation" – ponto de entrada da aplicação Java;
- "utils" – software utilitário do projeto.

Na diretoria "sql" existem duas subdiretorias, "routines", com os procedimentos armazenados, funções e gatilhos implementados, e "views" com as vistas. Nesta diretoria também estão presentes os 5 scripts autónomos já mencionados:

- create_tables.sql – criação do modelo físico;
- remove_tables.sql – remoção do modelo físico;
- insert_data.sql – inserção de dados iniciais no modelo;
- clear_data.sql – limpeza das tabelas do modelo;
- tests.sql – testes das funcionalidades dos mecanismos implementados.

# 2. Modelo de dados

Neste capítulo estão descritas as decisões tomadas durante a implementação do modelo de dados. O modelo de dados está na diretoria “src/main/java/pt/isel/model”.

O modelo de dados, previamente desenhado na primeira fase do projeto, foi implementado em Java com recurso à ferramenta JPA.

Para cada entidade do modelo, foi implementada uma classe Java, que representa essa mesma entidade. Cada classe tem como propriedades os atributos pertencentes à entidade, e alguns métodos de encapsulamento dos mesmos, os getters e setters. Nesta fase foram utilizadas anotações do JPA, tais como: @Entity, para definir entidades, @Table, para associar uma entidade a uma tabela da base de dados, @Id para especificar a chave primária de uma entidade, @Column para especificar as características de uma coluna associada com uma propriedade, etc. Cada classe também tem um método overrided *toString*, para melhor representação da entidade na consola, em modo debug.

Entidades generalizadas, como Client e RetrievedGpsData, são classes abstratas, estendidas pelas suas especificações. No caso dos clientes, Client é estendido por InstitutionalClient e por PrivateClient, no caso dos dados GPS, RetrievedGpsData é estendida por UnprocessedGpsData e InvalidGpsData.

Também foram implementadas algumas mudanças ao desenho do modelo de dados inicial:

- A entidade Client tem um novo atributo *dtype*, que pode ter como valores ‘PrivateClient’ ou ‘InstitutionalClient’; este atributo foi adicionado para o JPA conseguir dois clientes através da leitura da tabela *clients*;
- Todos os atributos *location*, que eram do tipo POINT, passaram a ser atributos compostos, divididos em *lat* (latitude) e *lon* (longitude), as coordenadas da localização;
- Foram adicionadas algumas restrições de integridade;
- A entidade UnprocessedGpsData tem um novo atributo *version*; este atributo tem o objetivo de o JPA conseguir garantir optimistic locking.

# 3. Camada de acesso a dados

Neste capítulo estão descritas as decisões tomadas durante a implementação da camada de acesso a dados. A camada de acesso a dados está na diretoria “src/main/java/pt/isel/dal”.

Uma camada de acesso a dados (DAL – Data Acess Layer) é uma camada de um programa que fornece um acesso simplificado a dados armazenados em algum tipo de armazenamento persistente, como uma base de dados relacional. Nesta camada são implementados padrões de acesso a dados lecionados nas aulas, entre eles Mapper, Repository e PersistenceManager.

## 3.1 Padrões de acesso a dados

A classe abstrata Mapper<T>, representa um mapper para uma entidade especifica, cujo tipo é passado como parâmetro. Este padrão de acesso a dados fornece um conjunto de operações CRUD (Create, Read, Update, Delete), sobre uma entidade.

A classe abstrata Repository<T>, representa um repositório para uma entidade especifica, cujo tipo é passado como parâmetro. Este padrão de acesso a dados fornece um conjunto de operações sobre coleções de entidades, como os métodos: *getAll*, *deleteAll*, *getById*.

A classe PersistenceManager é uma classe estática, que fornece métodos para gerir o contexto persistente, como por exemplo, criar/obter instâncias de EntityManagerFactory ou EntityManager. Esta classe contém o método estático *execute*, que executa uma transação, recebendo um bloco de código.

## 3.2 Implementação das funcionalidades

Com o modelo de dados e a camada de acesso a dados implementados, desenvolveram-se as funcionalidades descritas na fase 1 do projeto das alíneas 2d a 2l, em Java. Estas funcionalidades foram implementadas em procedimentos armazenados, funções, gatilhos e vistas.

### 2.d) private_client_procedures.sql

As seguintes funcionalidades dizem respeito à entidade PrivateClient:

- **insert_private_client**
- **remove_private_client**
- **update_private_client**

Para invocar estes procedimentos em Java, foi criado a classe PrivateClientRepository, que estende Repository<PrivateClient>. Esta classe contém os métodos respetivos para estes procedimentos: *add*, que insere um PrivateClient, *remove*, que remove um PrivateClient, e *update*, que atualiza os dados de um PrivateClient. Todos estes métodos recebem uma instância de PrivateClient como parâmetro.

### 2.e) get_alarms_count.sql

Esta função retorna o número total de alarmes para um determinado ano e matrícula de veículo, passados como parâmetro. Se uma matrícula não for fornecida, retorna o número total de alarmes para todos os veículos no determinado ano.

Como este método tem um parâmetro opcional (matrícula do veículo), foi implementado duas vezes. Uma na classe VehicleRepository, que estende Repository<Vehicle>, e que apenas recebe como parâmetro o ano do alarme, retornando o número total de alarmes para todos os veículos nesse ano. A outra implementação encontra-se na classe Vehicle, do modelo de dados, que também recebe como parâmetro o ano do alarme, mas apenas retorna o número total de alarmes para o determinado veículo, nesse determinado ano.

### 2.f) process_gps_data.sql

Este procedimento realiza o processamento dos dados GPS não processados, sendo chamado periodicamente.

Para implementar esta funcionalidade, foi implementada a classe GpsDataProcessor, que estende TimerTask, que é uma classe abstrata que representa uma tarefa que pode ser agendada e chamada periodicamente. Esta classe apenas tem um método, *run*, que executa o procedimento SQL, que realiza o processamento dos dados.

Inicialmente, o nível de isolamento escolhido para este procedimento era ***READ UNCOMMITED***, contudo foi necessário alterar o mesmo para ***REPEATABLE READ***, visto que o gatilho de inserção na tabela *gps_data* tem esse nível de isolamento.

### 2.g) generate_alarm_trigger.sql

Este gatilho permite analisar dados GPS processados, quando estes são criados, e gera alarmes, se o veículo estiver fora da zona verde.

Esta funcionalidade não foi implementada na aplicação Java, visto que já está presente na base de dados, sendo que o gatilho é chamado na inserção de dados na tabela *gps_data*.

### 2.h) create_vehicle.sql

Este procedimento cria um veículo e associa-o a um cliente. Se forem passados dados para uma zona verde, esta também é criada e associada ao veículo criado.

Esta funcionalidade foi implementada em quatro métodos distintos, dois destes que utilizam o procedimento armazenado desenvolvido na fase 1, e outros dois que apenas utilizam métodos da camada de acesso a dados desenvolvida. Para cada uma destas implementações, existem dois métodos distintos, ambos recebendo o veículo do tipo Vehicle como parâmetro, mas apenas um destes recebendo a zona verde, do tipo GreenZone, visto que a zona verde é opcional.

Todos estes métodos estão implementados na classe VehicleRepository.

### 2.i) list_alarms.sql

Neste script, é criada uma vista que lista todos os alarmes, juntamente com informação sobre o id do dispositivo, matrícula do veículo, localização e tempo.

Para representar esta tabela em Java, foi implementada a classe AlarmData no modelo de dados. Esta classe está anotada como uma entidade e associada à vista *list_alarms*.

### 2.j) insert_into_list_alarms_trigger.sql

Este gatilho permite, ao fazer a operação INSERT na vista *list_alarms*, inserir o registo e o alarme nas suas devidas tabelas.

Esta funcionalidade não foi implementada na aplicação Java, visto que já está presente na base de dados, sendo que o gatilho é chamado na inserção de dados na vista *list_alarms*.

### 2.k) clear_invalid_gps_data.sql

Este procedimento remove todos os registos inválidos da tabela *invalid_gps_data* que existam há mais de 15 dias, sendo chamado diariamente.

Para implementar esta funcionalidade, foi implementada a classe GpsDataCleaner, que estende TimerTask, que é uma classe abstrata que representa uma tarefa que pode ser agendada e chamada periodicamente. Esta classe apenas tem um método, *run*, que executa o procedimento SQL, que realiza o processamento dos dados.

### 2.l) delete_client_trigger.sql

Este gatilho é executado sempre que é feita a operação DELETE na tabela *clients*, e que permite, ao invés de remover um cliente, simplesmente atualiza o seu atributo *active* para false.

Esta funcionalidade não foi implementada na aplicação Java, visto que já está presente na base de dados, sendo que o gatilho é chamado na deleção de dados na tabela *clients*.

## 3.3 Optimistic locking

Neste capítulo estão descritas as decisões tomadas durante a reimplementação da funcionalidade *gps_data_processor*, utilizando a técnica optimistic locking. Esta funcionalidade está implementada na classe OptimisticGpsDataProcessor, na diretoria "src/main/java/pt/isel/dataProcessors".

Para alcançar este objetivo, foi reescrito o código do procedimento *process_gps_data* com as operações dos Repository. Para garantir que as instâncias de UnprocessedGpsData são apagadas com a mesma versão obtida pelo SELECT query, adicionou-se uma coluna *version* à tabela *unprocessed_gps_data*.

Após executar o OptimisticGpsDataProcessor com 100 threads, é possível observar que o optimistic locking previne erros de concorrência, devido a lançar a exceção representada na Figura 1.

```
[EL Warning]: 2022-06-06 05:51:51.319--UnitOfWork(578650563)--Exception [EclipseLink-5006] (Eclipse Persistence Services - 4.0.0-M3
 .v202203111216): org.eclipse.persistence.exceptions.OptimisticLockException
Exception Description: The object [UnprocessedGpsData{id=1, gpsDevice=1, timestamp=Tue Apr 12 06:05:06 WEST 2022, location=(18.0,9.0)}]
 cannot be updated because it has changed or been deleted since it was last read.
```

**Figura 1 –** Exceção lançada pelo OptimisticGpsDataProcessor

Como este processador insere na tabela *gps_data*, é utilizado o *generate_alarm_trigger*. Como este gatilho utiliza um nível de isolamento REPEATABLE_READ, a transação deste processador também deve ter esse mesmo nível de isolamento. Com o mesmo teste que corre 100 threads a executar o OptimisticGpsDataProcessor, é possível observar que o REPEATABLE_READ previne erros de concorrência, devido a lançar a exceção representada na Figura 2.

```
[EL Info]: 2022-06-06 05:51:50.378--ServerSession(381013035)--EclipseLink, version: Eclipse Persistence Services - 4.0.0-M3.v202203111216
[EL Warning]: 2022-06-06 05:51:51.139--UnitOfWork(1517766855)--Exception [EclipseLink-4002] (Eclipse Persistence Services - 4.0.0-M3
 .v202203111216): org.eclipse.persistence.exceptions.DatabaseException
Internal Exception: org.postgresql.util.PSQLException: ERROR: could not serialize access due to concurrent update
  Where: SQL statement "UPDATE vehicles
    SET num_alarms = num_alarms + 1
    WHERE id = vehicle_id"
```

**Figura 2** - Exceção lançada pelo OptimisticGpsDataProcessor com nível de isolamento REPEATABLE_READ

# 4. Aplicação de consola

Neste capítulo estão descritas as decisões tomadas durante a implementação da aplicação de consola. Desenvolveu-se esta aplicação para dar suporte às funcionalidades desenvolvidas anteriormente.

A aplicação está implementada na classe App, na diretoria "src/main/java/pt/isel/presentation". Esta classe contém apenas um método público, *main*, que é o ponto de entrada da aplicação.

Este método começa por chamar o método utilitário *createDatabase*, que cria as tabelas, vistas, funções, gatilhos e procedimentos armazenados, inserindo dados iniciais nas tabelas.

Após isto, são inicializados o processador de dados GPS, que chama o procedimento *process_gps_data* a cada cinco minutos, e o cleaner de dados GPS inválidos, que chama o procedimento *clear_invalid_gps_data* diariamente.

Finalmente, o main loop da aplicação é iniciado. Este ciclo imprime na consola o menu de operações disponíveis e espera que o utilizador selecione uma. Quando uma operação é selecionada, a função associada é executada, realizando a operação. Este loop corre até o utilizador selecionar a operação de exit, ou o processo encerrar.

# 5. Conclusões

Em suma podemos concluir que os objetivos definidos para a realização deste trabalho foram atingidos. Consideramos que desenvolvemos uma camada de acesso a dados adequada ao modelo de dados desenvolvido, cumprindo com todos os requisitos do sistema.

Durante a implementação, adquirimos conhecimentos relativos ao desenvolvimento de um modelo de dados, ao controlo transacional, camada de acesso a dados, padrões de acesso a dados e optimistic locking. Ainda adquirimos conhecimentos relativos à utilização do software DataGrip, que foi o IDE escolhido por nós para a manutenção e desenvolvimento da base de dados. Também aprendemos a desenvolver projetos Java com recurso à tecnologia de acesso a dados utilizada, JPA.

Este projeto proporcionou-nos a oportunidade de utilizar conhecimentos que viemos a adquirir nas aulas da unidade curricular, e durante o nosso estudo autónomo.

# Software utilizado

Segue-se uma lista do software utilizado na realização deste trabalho, juntamente com uma breve descrição sobre a sua utilização:

- **diagrams.net**: desenvolvimento de diagramas;
- **PostgreSQL**: linguagem utilizada para a implementação e manutenção do modelo de dados físico (base de dados);
- **DataGrip 2021.3.4**: ambiente de manutenção da base de dados e de desenvolvimento de código PL/pgSQL;
- **Java**: linguagem utilizada para a implementação da aplicação e da camada de acesso a dados;
- **IntelliJ IDEA 2022.1.2 (Ultimate Edition)**: ambiente de desenvolvimento para linguagem Java;
- **Java Persistence API (JPA)**: tecnologia de acesso a dados utilizada;
- **Git**/**GitHub**: controlo de versões e armazenamento do projeto num repositório;
- **LaTeX**: desenvolvimento do modelo de dados relacional;
- **Microsoft Word**: escrita do presente documento.

# Referências

[1] Walter Vieira. *SisInf_M1_Transacções(v6)*.

[2] Walter Vieira. *SisInf_M2_SP_Trig_Func*(v2).

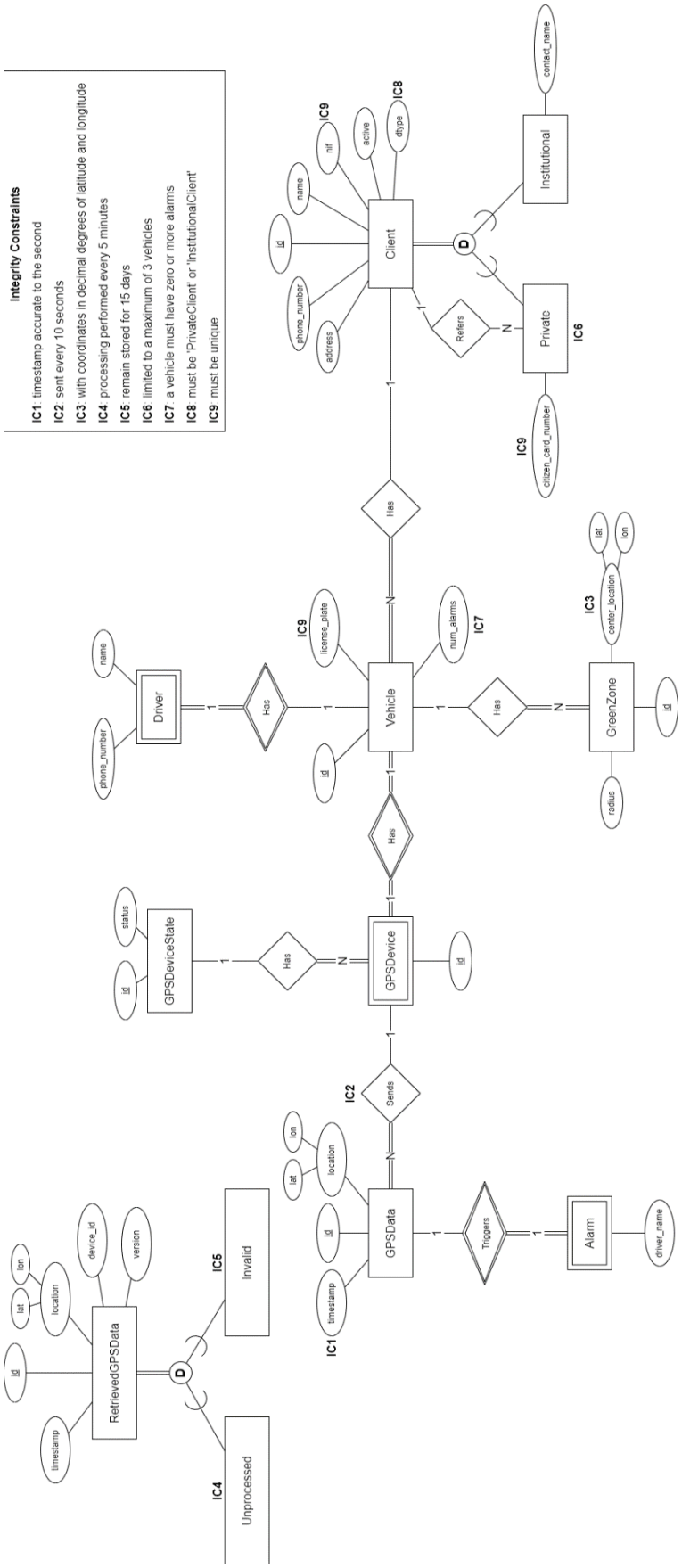
[3] *Documentation*. PostgreSQL. (n.d.). Retrieved June 4, 2022, from https://www.postgresql.org/docs/

[4] Walter Vieira. *SisInf_M3_Acesso_Dados(v2)*.

[5] Introduction to the java persistence API - the java EE 6 tutorial. (2013, January 1). Retrieved June 4, 2022, from https://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html

[6] Baeldung. (2020, March 21). Optimistic locking in JPA. Baeldung. Retrieved June 4, 2022, from https://www.baeldung.com/jpa-optimistic-locking

# A.1 Modelo de dados conceptual



# A.2 Modelo de dados relacional

InstitutionalClient(id, referral, name, dtype, phone number, nif, address, active, contact name)
FK: { referral } de PrivateClient.id

PrivateClient(id, referral, name, dtype, phone number, nif, address, active, citizen card number)
FK: { referral } de PrivateClient.id

GreenZone(id, vehicle id, lat, lon, radius)
FK: { vehicle id } de Vehicle.id

Driver(vehicle id, name, phone number)
FK: { vehicle id } de Vehicle.id

Vehicle(id, gps device id, license plate, num alarms)
FK: { gps device id } de GPSDevice.id

GPSDevice(id, device status)
FK: { device status } de GPSDeviceState

GPSDeviceState(id, status)

GPSData(id, device id, timestamp, lat, lon)
FK: { device id } de GPSDevice.id

Alarm(gps data id, driver name)
FK: { gps data id } de GPSData.id

UnprocessedGPSData(id, timestamp, lat, lon, device id)

InvalidGPSData(id, timestamp, lat, lon, device id)