Bachelor Project

University of Southern Denmark

IMADA

Microservice Mission Management Module

Linear Infrastructure Inspection

Advisor:
Prof. Peter Schneider-Kamp

Student:
Jacob Nielsen

Advisor:
PhD. student Lea Matlekovic

January 1, 2022

Contents

Chapter 1

Introduction

1.1 Acknowledgements

I would like to thank my advisor Peter for giving me change to make deep dive into the world of distributed systems, working with a real problem applying my knowledge from computer science theory. All related to the field of robotics, that I find very interesting.
Especially I want to thanks my other advisor Lea, that have survived a ton of questions and very long emails in her inbox monday morning. I really appreciate your patience and your understanding for bridging between the project specifications and the current stage of the project.

1.2 Project Description

I choose to work on this project as i saw an opportunity to contribute to a real world problem, applying my computer science knowledge. I found robotics and the system supporting behind very interesting. Microservice and distributed systems is very broad and big spectrum of tools of libraries. I have really enjoyed working on a tailored solution for this problem.
The first two chapters explains why we need this services and describes the existing code base and functionality. The rest of the chapters work with the problem. Designing and implementing the services defined describe how they integrate with the existing code base and meet the requirements.

Problem Statement

Below is the official problem statement:

Description
The mission management module is part of a mission planning application for linear infrastructure inspection and it serves as a data storage and delivery of virtual assets. It will store mission plans for each drone participating in the mission as well as telemetry data received from drones' sensors. Also, when demanded, the module will retrieve flight plans and tracking information from the database and deliver to the user interface, to image analysis services or to any other service requesting. Based on the drone's telemetry data, the module should estimate the drone's position and status.

Specific Tasks

- Getting to know the existing codebase and literature
- Design the database storage
- Implement the database storage
- Design individual services
- Develop individual services

- Integrate with existing services
- Evaluate your solution, propose future optimizations
- Write your Thesis describing the theoretical background and practical part in details

1.3 Abstract

Drones 4 Safety is a project developing a cooperative, autonomous, operating drone system to enhance transport safety. To support this project Linear-Infrastructure Mission Control (LiMiC) is being developed as a cloud service. From a UI, an operator can plan and control the continuos missions, inspecting the infrastructure. This thesis circles around the mission management part of LiMiC, serving as an API, that the UI can call. Designing and implementing relational databases with geospatial functionality is covered in both theoretical and practical details. The two databases is serving three dedicated microservices serving a REST API, needed the mission management software. Through the thesis the services is described in detail, handling questions about dependencies, internal architecture, performance and asynchronous-execution and database connections. Further the deployment aspect of microservices is described. Both the theoretical and practical parts is described in detail. All in all giving a working solution, a platform, to the problem statement that this thesis is all about.

1.4 Resumé

Drones 4 Safety, er et project som udvikler autonome drone systemer for at fremme sikkerheden i transportsektoren. En del af dette projekt, kræver en online cloud løsning. Her kommer behovet for cloudløsningen: Lineær-Infrastruktur Mission Kontrol (LiMiC). Fra en brugerflade kan en operatør planlægge og kontrollerer missioner og dronerne i disse. Denne bachelor opgave cirkler omkring LiMiC og the API som brugerfladen kan sende forespørgsler til. Igennem projektet bliver design og implementering af relationelle databaser med geo-funktionalitet teoretisk beskrivet og praktisk implemneteret i detaljer.
De to special-designede databaser bliver brugt til at servicerer 3 microservices som blotlægger hver deres REST API, med funktionalitet til mission- håndteringen. Igennem opgaven bliver disse services designet, udviklet og beskrevet. Biblioteker og andet software, sammen med den interne arkitektur, ydeevne og asynkrone kørsel- og database forbindelse bliver undersøgt, valgt og implementeret. Ydermere, bliver produktions-miljøer udviklet dedikeret til disse services. Både de teoretiske-og praktiske dele bliver beskrevet i deltajer. Alt i alt, leverer opgaven en platform, som løser de problemstillinger stillet i problemformuleringen.

1.5 Code

The code can be found on GitLab `Mission` project branch: `https://gitlab.sdu.dk/matlekovic/D4S/-/tree/MissionModule`
Further the whole branch is zipped with the thesis. The Services developed for this thesis is in the `missions`, `logs` and `estimator` directories.
It is recommended for the reader to have the code on hand.

Chapter 2

D4S - Drones for safety

2.1 Drone Inspection as a Service (DIaaS) Overview

Drone Inspection as a Service (DIaaS) is a service platform specially designed for the Drone4Safety project (D4S) [29], which is a project developing a fully autonomous system of self-charging collaborative drones for inspection of transportation infrastructures. The platform deploys its missions in a, so called ongoing operation, meaning the drones is deployed and essentially not managed before the mission is completed, without human intervention. The official goal of the project is to increase the safety of the European civil transport network through an accurate, frequent and autonomously inspections, both for construction workers and as a result passengers.
The project will produce both a software and hardware sector, where DIaaS, the software sector is designed in the layers:

- cloud services
- drone swarm
- network layer

The network layer are substantiating the communication between the cloud and the drone swarms utilizing a modular distributed message parsing technology.

The cloud serves a platform for mission control, navigation and monitoring through a web interface. The platform is designed in a more traditional concept frontend and backend, where the frontend consists of a web interface that relies on the backend and its clouds services. The backend consists of a modern microservice architecture constitute distributed system. The microservice are containerized using Docker deployed in a Kubernetes cluster.
The microservices include fault detection, mission control, swarm fleet management and data analysis, e.g. 3D reconstruction and image analysis.

All in all, a very loosely coupled system of specialized microservices services in their own encapsulated environment in containers, giving the benefits of modularity at hand both in the development process and further development and maintenance.
Below here an overview of the DIasS' goal.

2.1.1 DIaaS Backend

The backends dedicated microservices are divided by their functionality. This is introduced quite naturally but also gives some scalability benefits, as some functionalities requires more of a given resources, that then specifically can be adjusted between the services.

Image processing and machine learning

This service are handling computational image processing and machine learning. This is among other purposes to enable fault detection and construct 3D models from the pictures. The drone is logging live where it has taken a given picture but only uploading when allowed by a signal strength constraint.

Safety service

Safety service is analysing the data both from the drone and direct weather data to decide whether it is safe to start and or proceed flying the mission.

Charging service

Monitoring of the drone's battery levels and coordinating, planning and executing the command demanding the drones to charge.

Mission planning and scheduling

This service is acting as a command central dispatcher. Through this service, an operator can plan a given mission where the service care plan routes according to the current weather conditions and coordinate the time schedules for charging circumstances.

Wheather services

Responsible for gathering and communicating to drone sensors, the given weather information around the flying area to the drones. This enables services to better mission planning and scheduling.

Position services (Estimator Service)

This service calculates the drone's position in near real-time based on the flight plan, last received position updates, that the services get with other data like real position and velocity. This enables a relative given position when communication with the drone is not established. When the drones report to the cloud, the next estimate is based on that.

Data Storage and delivery services

The data storage and delivery are handled among several services. There is a service response from receiving telemetry data from drones and storing it in a suitable format, that makes sense and fit the analysis afterwards. This means that spatial data will be stored in PostGIS. These services also cover the flight plans, tracking information and inspections images. Both storage and delivery are on request.

2.1.2 DIaaS Frontend

The Frontend is a web based user interface, that communicated with the services using the HTTP protocol. Mission targets and UAVs is visualised together with the mission planning capabilities. An UI based on web-technologies makes it possible to port the application to different runtime environments, but also allowing an operator to access the web-based system from all supporting browsers. The interface would be role-based, defining privileges. An operator can choose the infrastructure to inspectthe specific targets and the UAVs (called sources). Inspect the missions route and initiate it and monitor it progress and status. After a mission, the tracked parameters can be visualised for analysis. The interfaces can be used for visualising the ongoing missions and UAV-positions and status'. (Note, there is few more details in [29] not directly relevant for the requirement of the thesis).

Chapter 3

LiMiC 1.0

3.1 Linear-infrastructure Mission Control (LiMiC)

LiMiC is an application for autonomous Unmanned Aerial Vehicles (UAV) infrastructure inspection. As described in the DIaaS section a fully managed backend (cloud service). LiMiC 1.0 Consists of 3 specialized services and 3 databases.

3.2 LiMiC Frameworks, Communication and UI Interaction

Communication in LiMiC 1.0 backend microservices are using the stateless HTTP protocol [40] by requesting their exposed API endpoints. In the first versions of LiMiC (in the monolith version 3.3), the services were in the earlier versions implemented using the Flask framework [15]. During the later decoupling to a microservice architecture, there was implemented a more performant replacement named FastAPI web framework across all services [88]. FastAPI is the backbone of the services' communication exposing the APIs endpoints as a REST API. The framework has asynchronous support which is crucial for the microservice architecture. E.g. this supports the concurrent computation of the shortest path, with calls from VRP to A* (3.6). These features are built-in with very little implementation needed [74]. Further, FastAPI facilitates the possibility for input validation, both for types and data-structure, building on top of the Pydantic framework. FastAPI generates automatically interactive documentation in both Swagger and OpenAPI. With the Pydantic features, this enables fully generated example testing [67].
Using the web interface with the 2D map (UI), users can select specific targets (3.5) such as power towers or railway lines and then start a computation of the order of visits and from this information generate the waypoints. The web interface shows the UAVs and towers on the 2D map. The UAVs flight- and mission status is visualized appropriately.

3.3 From Monolithic application to Microservice Structure

The preliminary system design of LiMiC was developed using monolithic architecture. Such system decisions impact an applications quality, performance, maintainability and usability [28]. When starting developing an application the business perspective and the modern work-schemes such as e.g. SCRUM [59] (or other agile techniques) often dictates the development of an Minimum Viable Product [27] (MVP), hence not taking on the extra complexity as e.g. microservice architecture introduces. This approach becomes unsuitable when the number of application features grows fast (Often a list provided by the same SCRUM/Agile stakeholders). Hence, if the need for scalability, maintainability and extendability occurs the architecture of the application should be reconsidered [28].
Monolithic applications are in general more tightly coupled and have a big codebase resulting in more complex applications. The development does not require architecture planning as such, since the applications are developed and easily deployed as a single instance and run "as is". Further, it is easy to horizontally scale such an application by running multiple instances behind a load balancer [28]. Testing monolith application is not complex as testing distributed systems, as end-to-end tests can be performed using test-suite (See how we test our microservice in chapter 6)
Where the big drawback comes is, as already hinted, when the code base grows and many developers contribute, with

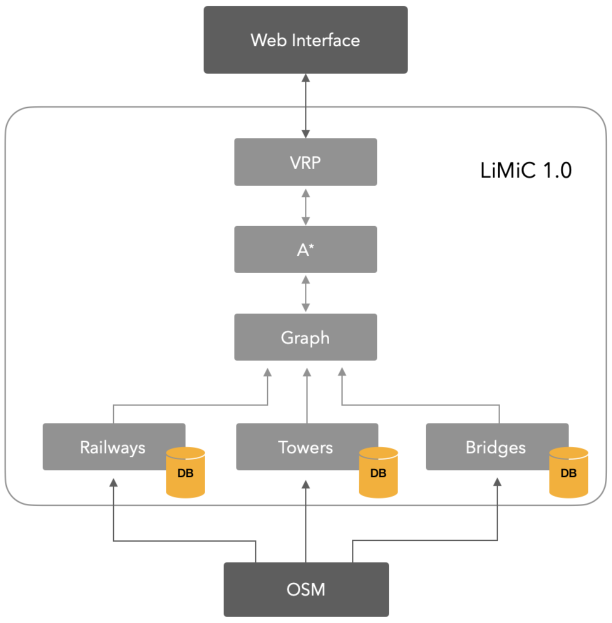


Figure 3.1: Microservice Architecture of LiMiC 1.0

components that potentially are tightly coupled. This results in a complex and difficult to understand application. Implementing smaller updates and fast agile development is becoming hard. Each update requires the entire application must be redeployed, making it hard to encapsulate possible unwished side effects. Bugs in a module can potentially crash the whole system. This makes the necessity for extensive and time-consuming manual testing, ensuring that each affected area of concern works correctly. Especially the integration-tests needs to be verified or rewritten. The easy scalability of these applications is often not the most optimal one, as the load is not equally distributed among all the modules. The modules have different resource requirements, coursing unwanted CPU and memory consumption making an overall bad utilizing of resources resulting in worse scalability. Monoliths are complex to change. Implementing new frameworks or technology would lead to a total reimplementation of the codebase [28]

A monolithic architecture can be a good starting point when the goal is a simple product. However, if the benefits of distributed systems are known to be needed up front, or when the complexity of the codebase grows and slows down development, the architecture should be reconsidered.

3.4 Architecture

The application is constructed as a Microservice architecture. This means that the complexity of the system is relatively lower, as it is divided among multiple small dedicated services. This also open for a frame of producing small organized services, that can support a rapid/Agile development cycle, that supports both simpler testing and high maintainability [28].

Microservices have their own runtime environment and often resides in their own virtual container, which is evidence of how clearly the services are loosely coupled and that given communication need to be through a given protocol. Microservices are deployed independently totally supported by the architecture, making independent and automatic scaling depending on application load. This fact opens together with a common communication protocol, for the use of different technology stacks, including programming language and hardware, giving overall flexibility for a given use case. This of course comes with a trade-off, as microservices is a distributed system, and hence handling of fallacies as latency, network throughput etc. to be handled. The impact on the system needs to be analysed against the needs and requirements. This introduces an extra level of complexity.

There are contrary opinions to which architecture to pick in the early stages of development. There are opinions advising to developing an in a monolith architecture and then migrate to the microservice architecture later [32], where others advise starting directly with the microservice architecture. There can both be complexity in starting with microservice architecture and later migrating from a monolithic one. There is a development bottleneck in any case and in practice, it is probably a resource, business and scalability question.

3.5 Data Sources

The data source for power towers and power Lines is Open Street Map (OPM) [44], which consists of open-source crowdsourced data. This means that the validity of the data is not guaranteed, which means that the accuray or whole placements could be imprecise or tampered [31], but it seems to be satisfactory for the current needs. There exist validated data in some European regions. Denmarks Government-owned company Energinet have publicly accessible data for 400 kV, 150 kV and 132 kV power lines in Denmark [18]. The danish instance Kortforsyningen (Map Supply) provides for both power lines and high voltage power towers. Moreover they also provide data on antennas that could be relevant for safe route planning [14].
The OPM database contains coordinate and id-number for the power towers. The given data is extracted in the services using the Overpass API with custom queries citeOverpass. The Services using this source are Tower and Railways.

3.6 Services

Below here is given a brief overview of what functionality the existing services provide. This is the services that this thesis would use and integrate with, ultimately becoming LiMiC together.

Tower Service

Tower Service extracts the power line data, as described in data sources and saves it in a database. The service uses the MongoDB Atlas database [39], which is a simple implemented database. Further, the database provides a multi-cloud database service. This means it is a very resilient, scaleable and performant database system. Mongo stores data in JSON-like documents. 2-dimensional Geospatial capabilities are supported [38] making it possible to perform a geo-based search, which is used for finding indirect towers' needed for determining edges in the construction of the graph (Later we would see it it not enough, for the services develop through this thesis). Likewise, the Power Lines are stored as object collections containing unique ids and an array of unique ids representing the nodes the line is connecting. Further, these collections also describe the information of the given line, the number of cables, frequency and voltage. Note that, Nodes can both be towers and line intersections, hence there needed to be a data structure describing this separation. They have the same data representation as towers. The power line data is used for describing direct neighbor relationships. A neighbor is not only the closest tower(s), they also need to be a part of the same power line. This allows the creation of a graph with a higher edge-penalty between indirect neighbors, causing the UAVs to follow the power lines infrastructure as close as possible which meets the UAV flight regulations [28]. The tower service exposes two endpoints. One for the tower data sand another for power lines.

Railways Service

This service extracts the railway's locations as described in the data source section. The data is saved as explained in the sections above, saved using the MongoDB database.

Bridges Service

Bridges services extract geographical polygons around the bridges and save them to the MongoDB database. The polygons are stored in objects containing a unique id and an array of unique id referring to nodes. Nodes are stored in the same database containing the node's location. The bridge's location can be combined with the power lines nodes to create a graph and enable UAV's to get a path to a bridge along with the infrastructure.

Graph Service

The graph service is responsible for building the graph from the data requested from the tower-, railway and bridges-service. Graph structure is used to store path data, meaning the nodes in the graph represent power towers, containing the id and the geographic coordinates. Edges in the graph contain the distances between neighboring towers which is used to describe the costs of flying between the towers x_i and x_j. Graphs are built using the NetworkX Python package [41]. The graph is created once and is used by A* pathfinder to determine the shortest path between the set of inspection targets and UAVs.

VRP - Routing Solver Service

Finding an optimal route for the vehicle routing problem (VRP) is NP-Hard [45], hence the size of the problems that can be solved optimally is limited. In LiMiC, Google OR-Tools [20] are used to nearly optimally determine the solution to the VRP problem sizes relevant in our situations. The problem is defined from the set of targets (e.g. power towers) from a set of sources (UAVs). The VRP service builds a distance matrix using shortest path computations from asynchronous requests to the A* service. The solutions is a path for each UAV as a set of waypoints in the form of latitude and longitude coordinates.
There exists optimized algorithms and strategies for path planning, actually researched for the purpose of this use case. Possible to create path to europa in a few seconds [19].

A^* Path Finder Service

A^* service computes the shortest path using the A* algorithm, implemented in the NetworkX library [41], based on the data stored in the graph, from graph service. The paths are computed by receiving each combination of selected UAVs- and targets-locations received from Route Solver Service. The service returns the shortest path between a target and a UAV, with the total path distance as well as the distance between each path segment each with corresponding node locations. The result is returned to the vehicle's scheduler.

Deployment Pipeline, DevOps tools

To make an easier continuos development, automatic building and testing a devops pipeline is set up. GitLab is used for both repository, enabling easy collaboration and version controlling. GitLab has built-in tools for continuos-integrations(CI) and deployment(CD). The application is also deployed using Gitlab, where the pipeline is configure using each services `gitlab-ci.yml` file.
The the idea of microservices is to manage a relatively little independent and isolated code base, that independently can be scaled and managed. To be enable that, a containerization technology is used. Docker is used to encapsulate each microservice individually (using their dedicated Dockerfile) with its own software-package, runtime, system libraries and settings [28]. Every push to the GitLab repository trigger a Gitlab Runner event, that executed the scripts defined withing th `yml` file, creating a cluster based on all services's files.
The automatic deployment scaling and management of the containerized applications is handles using Kubernetes (MiniKubes Kubernetes cluster for now [28]).

So Why Microservices?

The short answer is performance. As Microservices comes with great scalability, modularity and encapsulation both in deployment and in functional seperation in between services, it does not out-weight the performance parameter in practice. But as microservices introduce more complexity during development and deployment, they comes with great benefit. The microservice architecture is much faster than the earlier monolithic implementation [28]. The microservice implementation were ony slower in one situation: when there is chosen two, four or eight UAV's for one target. An unlikely use-case in practice. Microservices allows us to scale up on services the computationally hard problems (heavy load in general), or services with a lot of traffic/load. What must be tested is that the containerization penalty of virtualization of multiple processes and their communication in-between overhead does not eat up the other benefits. Which they don't do in this case.

Chapter 4

LiMiC 1.0+

The project, vision and existing code base and its evolvement is described. Now the focus is the solution to adding a mission management module to LiMiC1.0. With the specified requirements, the existing code base and deployment strategy in mind. The rest of the thesis would design and implement three microservices together makes up the mission management module. Everything building upon the existing LiMiC1.0 is called LiMiC1.0+ in the following chapters.

4.1 Choose a Database System

Choosing a database system is not always a trivial task. It requires insight in the requirements of the system. Depending on the use-case of the application, a evaluation of the actual utility of the database musts be analysed. Can we optimize the use of the databases functionality, by moving logic from the application to the DB and optimize the overall performance or lower the complexity? This can be because of the specific uses case. E.g. a search that is optimized in the database data structure storage (geospatial data) or when using a relatively slow programming language, as Python and a fast database.
In the Drone4Safety (Specification of the Drone Inspection as a Service platform (M6)) [29] the use of a PostGIS Database is specified for the services in LiMiC. Anyways, it is decided to question this choice against the current and future requirements for the project in order not to make rash choice. The most common different flavors of systems will be evaluated against LiMICs use case below, mainly against the relational (RDBMS) scheme.

4.1.1 NoSQL

NoSQL or as it called now Not Only SQL are non-tabular databases that exist in min flavors depending on their data model. The most common onces is key-value, wide-column and graph. Relational databases introduces a degree of unpredictability. Relational databases rely on the SQL query language, that rely on how well the SQL compiler does. Performance can be like a black box, because of the many factors impacting it and affecting how quick queries will return as well as how many concurrent queries that run at the same time
In this evaluation the focus is on Amazon AWSs DynamoDB, key-value database that, is considered one of the best and easiest NoSQL Databases. In a relational database, a `JOIN` or a `GROUP BY` may work well when there are few rows, but how do they scale? Actually these operations gets slower and slower by the size of tables grow.
Many databases are difficult to performance test. Often its about creating the right indexes and plan capacity, but this is different scenario when we go out of the testing environment, and the application hits a real scale. DynamoDB forces you to design a data model that will scale. This means that the performance is not a black box and the database will scale. The query-time would be around the same regardless of load and queries do not impact each other [1].

Scaleability: NoSQL versus RDBMS

RDBMS is a really powerfull tool and is very flexible for changes, supports OLTP (Online Transaction Processing) and OLAP (Online Analytical Processing) functionalites [1], but the scale does remain as the party breaker.
While computing SQL joins, data is read and compared from more than one different tables. This introduces at least a time complexity of linear time. This is a CPU extensive task that will require a database scale when the amount of data is growing, which introduces the next problem: It it hard to horizontally scale a relational

database, making vertical scaling the next solution. Vertical scaling meaning increasing the hardware resources on the existing machines, that will have a limit. Horizontal scaling for RDBMS tricky, and will be with big cost. e.g splitting a table accross multiple machines would mean that joins require a network request, introducing more CPU time, round trip time and to rely on the network connection. Another significant barrier it that `SQL` queries are unbounded, meaning there is no limit on the amount of data a developer is able to request in a single query, opening a potential for locking up many database resources.
So how do NoSQL solves these obstacles? First `JOIN`s are not allowed. In RDBMS we normalize our entities across our tables, essentially voiding repeating data. This comes with the two important benefits of storage efficiency, as we do not repeat data, and data integrity as its simpler to ensure, because when the data is updated in one row/table, all other places refer this row. The consequence of this, is that the data is very divided and here we have the joins to assemble the data, introducing much, but costly, flexibility in our query-access to the data, meaning we don't need to consider how we access the data. This means that NoSQL databases needs to consider the benefits of having joins:

- Flexible data access:
  Totally avoided, becauses of the database structure forcing you to consider how data is both reade and writed. The overall idea is that you don't reassemble data on the go, but pre-join data by structuring in the way you will read it.

- Data integrity:
  This is a big tradeoff for some use cases. The data integrity is changed to be expected to be handled by the application. Data bay be needed to be denormalized and duplicated in the database. This means that updating data, could mean updating multiple records.

- Storage
  Today this is not considered a big problem anymore. Companies would probably gladly buy efficiency for the price of memory.

The horizontally scale

Is possible in NoSQL because of the pre-joine (when designing schema) contrary to the flexible syntax in RDBMS that navigates in shattered data, making it hard.
Data can be splitted into data segments, smaller accesable chunks, where the query then can be preformed. In DynamoDB and Cassandra these are called a partition key.
The basic concept in NoSQL is to think about the database consisting of a hash table, where the value of each key is a b-tree. Then by using the partition key to define the keys in the hash table data can be spread our on an unlimited nodes. The use of a hash table relies on implementing probability algorithms, that can help distribute the data evenly. The cost for this, is that all queries must include the partition key, but it could simply be an ID attribute. This sharding mechanism is what powers the scale without performance degradation [1].

DynamoDB bounds

The important idea here is bounding the queries. Finding the value of the hash table is a quick operation and traversing a B-tree is also efficient, but this af courses doesn't scale naturally. Therefore DynamoDB imposes a limit of 1MB on a query, returning a "last evaluated key"-id allowing pagination controlled by the client. This is all the requirements for DynamoDB to offer the guarantee of single-digit milliseconds latency on a query on any scale. This is with constant time, O(1), for calcualting the partition key and then O(log n) (n= size of collection) for finding starting value for read. This is of course not without a trade off. DynamoDB does not offer any aggregation operations. Next the client controlled pagination does not scale (it opens for the client to destroy the concept) [1].

NoSQL is not the holy grail. It requires a lot of insight for the use and growth of the application and the use of data-compositions. It is good for big data purposes, which is also its biggest customer. Last a very important but small fact, it the concept about data-integrity. The use case i LiMiC is safety and UAVs in the air and we can't rely on a database that doesn't guarantee the data-integrity across all data when create or updated, at least not for some of it.

4.1.2 New SQL

As concluded above, NoSQL falls short for some ues cases, including ours. This is primarily because of the database system not guaranteeing the `ACID` properties but instead implement the properties of `BASE`. It is a general concern that this new paradigm of database systems is not compatible with the old systems. New SQL could be the solution. It is a relational database with the scalability properties of NoSQL and still providing the ACID properties. NewSQL allows easy horizontal and vertical scaling, by implementing a distributed database technology using technology used in cloud computing and distributed applications. This is done by following a three tier architecture of three layers:

- Administrative layer
- Transactional tier
- Storage Tier

NewSQL's high performance is a result of keeping all data in RAM (Random Access Memory). The scalability is reached by using partition/ sharding and replication in a structure that in general means queries do not have to communicate between multiple machines. The required information is orchestrated by a single host. All this, makes the database system it possible to handle Big Data across many machines efficiently. [76]

NewSQL supports the ACID properties

The schema is a combination of SQL and NoSQL witch the catch of a complex queries.
Google Spanner is based on NewSQL providing a globally distributed scalable database. The database uses sharding technique to split the data across many Paxos state machines. The Paxos machines is then used to solve the consensus among multiple results (of data). In this specific system, the database is globally available, which requires the data to be replicated, to achieve a low latency. Data is versioned automatically. This is done by consider the geographic locality and hot key analysis (What is accessed most, like a CDN in small format). Spanner automatically reshards and migrates data dynamically, making a load balancing effect. Quering in Spanner is done with the `SQL` query language[76].
All in all NewSQL is current perfect candidate for a database providing consitentcy, scalability, speed and availability. It checks all properties for a Big Data OLTP application, that we would like in LiMiC. The database system is still on a infant stage and hence not applicable for LiMiC currently.

Relational Database Postgres System with PostGIS Extension

This leads us back to the traditional RDBMS. A requirement we haven't considered in this analysis, but mentioned in the LiMiC description, is the use of geospatial data, in 3D - making a `GIS` database a requirement. The choice is a `Postgres` [49] database with `PostGIS` extension installed [46]. This is chosen, as they are open source, well documented and very fast systems. PostGIS uses `QuadTrees` for spatial indexes [48].

4.2 LiMiC1.0+ Database Design

4.2.1 Design of Relational Database Schemas

When designing a schema, we want to avoid creating anomalies. The main anomalies that is encountered are:

1. Redundancy
   As stated, unnecessary repeated information in several schema tuples
2. Update Anomalies
   Changing information in one tuple, but leave the old/invalided information in others. This is, as explained in out analysis of database systems, potentially allowed in NoSQL databases introducing a threat to data integrity. We could be careful and update all tuples, but this is considered bad design.
3. Deletion Anomalies If a set of values becomes empty, we may lose other data as a side effect.

Avoid anomaliess

To avoid anomalies we decompose our relations if needed. A decomposition of a relation R means splitting the attributes of R construct the schemas of two new relations. The goal of this decomposition is to replace relations to others that do not produce anomalies. A relation $R(A_1, A_2, ..., A_n)$ are decomposed into to relations $S(B_1, B_2, ..., B_n)$ and $T(C_1, C_2, ..., C_n)$ such that:

1. The union union of S and T equals R: $T = S \cup T$
2. S and T respectively equals their projection of their set of attributes onto R:

$$S = \pi_{B_1,B_2,...,B_m}(R) \text{ and } T = \pi_{C_1,C_2,...,C_m}(R) \text{ [22]}$$

4.2.2 Surrogate or Natural keys

We must make a choice to use natural or surrogate keys in our relation. The natural keys will always stil exist and be functional in theory. In this design we use a surrogate key, to have one unique identifier for each tuple. A key is an attribute that all attributes in the functional dependency relies on, there can be multiple keys (naturally).

A primary key (PK) can influence som implementation issues such as how the relation is stored on disk (due to indexing). This is highly dependent on the data type chosen, which also to some extend effect query-performance. We often see the use of Integer or Big Integers as the standard auto-incremental integer-sequence id in implementations, but in practise there are several things to consider [91].

First of all having an integer sequence, makes it possible to reason about how big the tables are and extracting the keys opening a security hole. In general PK's should not be exposed in production environments, but this are quite common. It very easy as it is just creating a request and inspect the id returned, hence it is easier for guessing bots to utilise which defeats the idea of starting the sequence with an offset. Integer values also includes the possibility of encounter an overflow. An integer sequence also involves having identical across tables, which could course problems, because a foreign key (FK) are semantic and not just technical (natural). This could be a problem whene using sharding or similar. Instead the type Universally unique identifier (UUID) could be more fitting. UUID is a hashing mechanism. As they are not the most efficient database indexes, they guarantee 2.71 quintillion generations (uuid) before there is 50% change of a collision [34]. Hence they are a good choice in a staging environment (is used extensively in production also) and further do not expose the number of tuples, as it is not possible to enumerate values. Further we are almost guaranteed unique keys across tables and we do not need to consult the database before creating an id Having sequence as the "inner" key and then map everything in another table to an outer key constructing a faster inner index for the database more secure and flexible to the outside world. This construction can turn into a big problem when merging databases and all the PK's collide. [91]

Taking it all into consideration, the UUIDs would be the best overall choice and easy to manage across tables and in a distributed environment as they are robust to changes in the current staging environment also taking the performance performance into account. The primary keys are mainly implemented using uuid's but not in the drone-relation. This is because of the VRP service is not implementing this key strategy. The drone relation relies on simple string to bridge between the services for now.

4.3 High-Level Database Models

4.3.1 Design Principles

When designing a database model, there is a few simple principles that you should keep track of, that would prevent creating obvious anomalies. First create a model that reflects tha specifications of the application. Avoiding redundancy, is as already noted important. Further redundancy can arise when creating diagrams, for example when creating relations, that could introduce repetition and update anomalies. In general the design should be as simple as possible and rather keep representations in the same schema if possible, instead of creating new special-case schemas. As relations helps us connect the data (Entity Sets/UMLs), they should be added carefully. We do not want a situation where connected pair of entities for one relationship can be deduced from one or more others. This can introduce redundancy, update - and deletion anomalies. We can prevent this by projecting a set entities onto another set. (e.g. can derive a relationship by others). These principles will be applied in the database design for LiMiC1.0+.

4.3.2 Unified Modeling Language (UML)

UMLs is used to model the databases. This concept contains offer much of the same capabilities as the E/R models do. UMLs represent the entity sets almost as classes and in general in a more compact notation. They are used here because they are easier to to grasp and change. Theoretically they have different terminology. A binary relationship in E/R are an association in UML etc.
As notation and the true strictness to the correct theoretical formats vary a lot, note that the notation from Database Systems Book is used [22].

Missions Microservice

Below here is the UML diagram for the Missions Service. The aggregations- and composition relationships is added between the relations. The labels are kept with the forms, this is intentional, for readability. The (FK) means that this attribute is primary key in another database. These are quite important in this system, as PKs are used to make the queries needed. With a mission identifier, it is possible to derive all relevant data for a given mission. The data-interaction can be seen in appendix: 9.10

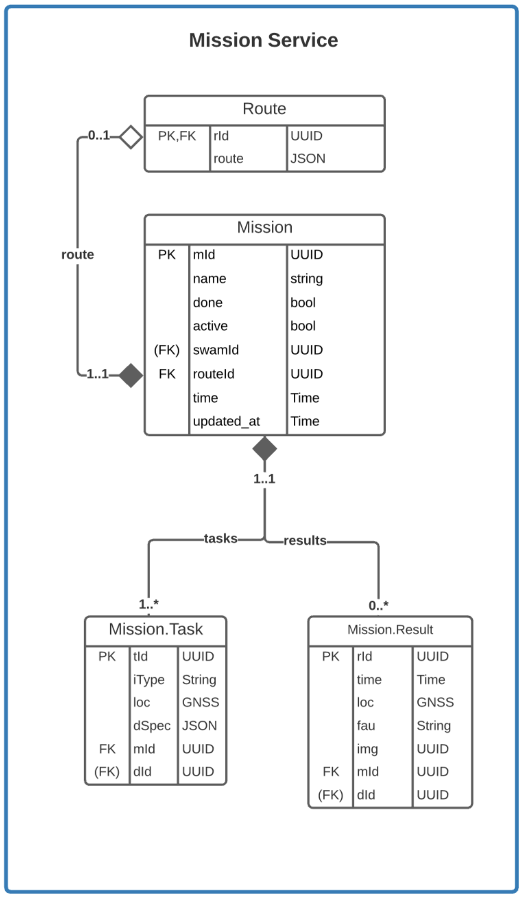


Figure 4.1: UML Mission Service Database

Drone/Logger Microservice

The UML diagram for the Drone Logger Mission. When the UI and VRP relies on the data from this service. It is possible to change the drone id to UUID, instead of the string.

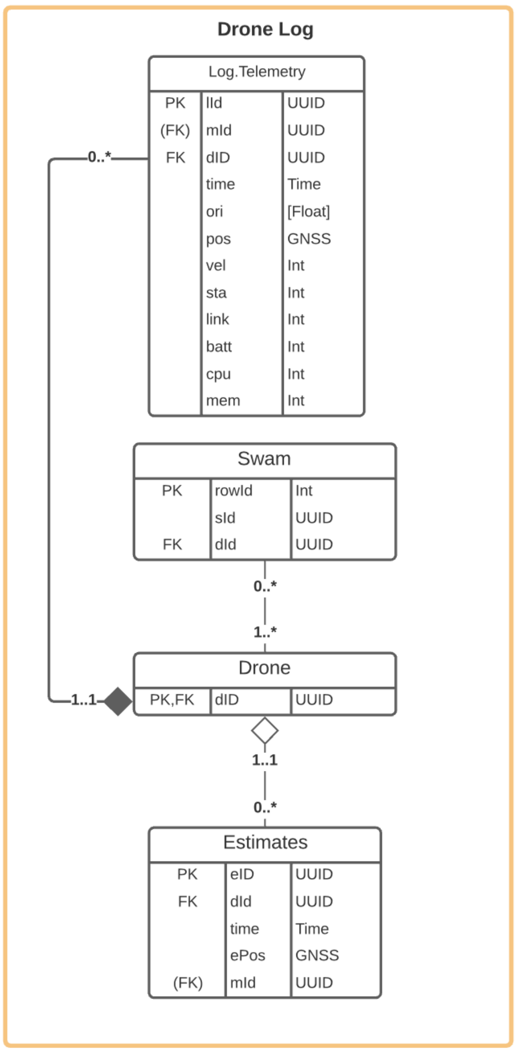


Figure 4.2: UML Drone/Logger Service Database

4.3.3 Missions Service Schema

The database in Missions Service consists of 4 UML classes.

Route

The Route class (schema) is responsible for the structure around the generated routes from the VRP-service. The class contains the attributes ‘id‘ to uniquely identify a tuple and ‘route‘ referring the given. Routes are encoded in raw Javascript Object Notation (JSON). This is both format the routes are received in from the VRP and the UI expects. Hence this format preserves multiple request for the same route with the exact same data.

The Route class is associated with Mission Class (see below) and can have 0 or 1 Mission, hence the rId is primary key (PK) for route and serves as a foreign key (FK) in Mission.

Mission

Mission class is containing meta information out the mission and associate the tasks and results.

mId `[UUID]` A simple boolean denoting if a mission is done.

name `[String]` A user-defined name associated with the mission.

done `[Boolean]` A simple boolean denoting if a mission is done.

active `[Boolean]` A simple boolean denoting if the mission is active.

sId `[UUID]` Refers the designated (selected drones) in a swam. (FK) in the diagram denotes that this is a primary key in another database in the distributed system.

rId `[UUID]` The route id for given mission. This makes it possible to request the route again to display.

time `[Time]` Timestamp of creation.

updated_at `[Time]` A logging feature to keep track of changes in the tuple. This can be used to keep an overview and used to investigate when certain values was updated.

Relationships:
Mission has a one-to-many association with Task class, as a mission would also contain at least one Task. Likewise, there is a zero-to-many association. This is explained by a mission can have zero results, as any mission starts with none and inspections that do not have any faults on the infrastructure (in this use case) would end the mission with none.

Task

Task class is responsible for holding information and specification for each way point that a route consist of.

tId `UUID` Per request in D4S specification each task is uniquely identifiable.

iType `String` Describes which type of inspection there on this way point. Could be e.g. tower or rail ways. This is mainly to tell the drone what is has to do with. This is a string in order to be flexible. The UI must restrict the allowed values. The type could also be changed to a enum type and then the database could enforce the pick.

loc `WKB` The GNSS geolocation coordinate, describing the geographical position in 3 dimensions. This is stored in Well-known Binary Format (WKB).

dSpec The GNSS geolocation coordinate, describing the geographical position in 3 dimensions. This is stored in Well-known Binary Format (WKB).

mId `UUID` The association / relation (in database lingo) to the mission.

dId `String` Which drone that is associated with this. This is dictated by the VRP-result. In the diagram (FK) denotes that this is a primary key in distributed system, but not in this database.

Relationships:
Have a composition relation to Mission class (1..1). Belong to exactly one mission

Result

Result class is responsible for containing the information for the found defaults during the mission.

rId `UUID` routes is per request in D4S specification each task is uniquely identifiable. This is a UUID.

time `Time` Timestamp for creating. This can be used to refer the fault detected with a given time, but also to analyse the path of the drones.

loc `WKB` The GNSS geolocation coordinate, describing the geographical position in 3 dimensions. This is stored in Well-known Binary Format (WKB).

fau `String` fault component type. String describing the fault type detected.

img `UUID` An UUID id type referring an image uploaded in object storage, a separated dedicated storage (Not a part of this implemenetation).

dID `String` Per request, it should be possible to make q query for the drone that inserted the result.

Relationships:
Have a composition relation to Mission class (1..1). Belong to exactly one mission.

4.3.4 Drone /Logger Service

This service contains the classes Telemetry, Drone, Swam and Estimates. All classes having to do with the operation of the drones and used to support the missions. The are in this class as the information needed in the UI can be collected from these tables without requesting information from other services.

Telemetry

Telemetry Class contains the drones reported statuses from the on board sensors. This data is used to monitor the drones, but also the possible for later analyzing its route.

IId `[UUID]` Telemetries should be uniquely identifiable. This makes an easy referable to "a point in time" for a specific drone with all it's on board statistics.

mId `[UUID]` Telemetries should be individual identifiable, by a single attribute.

dId `[String]` Telemetries should be individual identifiable, by a single attribute.

time `[Time]` Timestamp for the telemetry.

ori `[Array<Integer>]` Orientation of the drone at the reporting time. Containing roll, pinch and yaw represented in an array type. This type is in conflict with the traditional relational model containing atomic types, but this is a type in Postgres. Further this is always array constructed at length 3 and therefore the choice instead of another table.

pos `[WKB]` The GNSS geolocation coordinate, describing the geographical position in 3 dimensions. This is stored in Well-known Binary Format (WKB).

vel `[Float]` The velocity of the vehicle. This is a float as most units and precisions are supported.

sta `[String]` Flight status for the drone. This is a string currently as there is no definition on what it should be.

link `[Integer]` Signal strength for the drone. A integer describing the connection strength. Integer is chosen as its resolution enough and saves a little memory.

batt `[Integer]` Battery level for the drone, described by a integer on same argument as in link.

cpu `[Integer]` Status for on board central processing unit (CPU). An integer describing the load of the processor.

mem `[Integer]` Memory status for the on board memory, describing an integer percent

Drone

Drone Class is containing the drones. A Drone is very simple in this stage of the project, but with this class it is very easy to implement new attributes.

dId `[String]` A drone, is in this version only consisting of its identifier. This is a string type for the reason of integration with the VRP service. Using the same arguments as in the Key Section, it would be more optimal with type UUID. Hence the IDs could also be generated securing no-clashing (with a certain percentage) in the primary keys.

Swam

A swam of drones is allocated to a mission. To keep track of these allocated drones, we have the swam. Be aware that swam is called `alloc` in the specification [29], and is renamed here for higher readability in documentation and code.

rowId `Integer` SqlAlchemy enforces a primary key in each defined relation. Probably this is to enforce the implementer to decide on the indexes used. This is a simple integer, as it will never be referenced. In reality we don't need at unique identifier, as this relation always will be queried using the swam id (sId) that is a unique attribute in mission (This is a result of the way missions are generated)

sId `UUID` Swam id, a shared identifier on all tuples that is part of that swam.

dId `String` drone id, identifying the drone. This is a foreign key association with the drone class

Estimates

As the drones to not have a continuos connection to the system, we are implementing an position service (REF TO INTRO). The estimated positions is calculated by the Estimator Service (REF), but is stored in Logger Service. This is a design decision to collect all data that potentially should be joined or requested together, to be in the same database.

eId `[UUID]` Identifiers for uniquely describing the estimates. This is a requirement from specification

dId `[String]` Identifier for the given drone.

time `[Time]` Timestamp for the estimate creation in database.

ePos `[WKB]` The GNSS geolocation coordinate, describing the geographical position in 3 dimensions. This is stored in Well-known Binary Format ([WKB]).

mId `UUID` Mission id for the mission the estimate is generated during. It is possible to extract this information from the Telemetry table using the drone id and the timestamp. But as the queries properly will be the other way around, we would always make a query on these estimate tuples, using the mission id. The other solution will require to first query the first and last timestamp in the telemetry table and then estimates, but there could potentially be missing a few tuples in the result, so we circumvent these issues.

4.4 Normalizing Relations

As explained, to avoid anomalies we normalize our relational schemas. We are using normalization theory, that builds on the theory from functional dependencies. Normalization theory defines six normal forms (NF), each defining a set of dependency properties that a schema must satisfy. This allows each NF to (at most times) make a guarantee on potential anomalies. The key driver for this is to bring down the amount of redundancy. There is six NF's, but often only the first three is considered. We consider the first 4.

A set of X attributes in relation R:
Superkey:
K is a superkey if an only if it determines all other attributes [21]

Candidate key:
L is a candidate key if and only if its a superkey, but none of its proper subsets is a super key. If Y is a candidate key if and only if: $X^+ = R$, but for any proper subset $Y \subset X, Y^+ \neq R$

First Normal form (1NF)

- Only single values permitted at the intersection of each row and column (no repeating groups)
- Construct separate relations for each set of related data
- Identify each set of related data with a primary key

Second Normal form (2NF)

A table is in second normal form, if there for every non-trivial functional dependency $A \rightarrow B$ either:

- A is not a proper subset of any candidate key
- b is a key attribute

Third Normal form (3NF)

A table is in third normal form, if there for for every non-trivial functional dependency $A \rightarrow B$ either:

- A is a superkey
- b is a key attribute

Boyce Codd Normal Form (BCNF)
A table is in BCNF if there for every non-trivial functional dependency $A \rightarrow B$

- A is a super key

When normalization you take the natural keys into consideration and do not look at surrogate keys. Actually at this point it is wrong to think about the surrogate keys. Hence they are left out for these analysis, but they are added again in the implementation.
We are going to document that each of the relations hold all normal-forms our document the decomposition. This is a tedious process, but is essential in good database design.

4.4.1 Normalizing LiMiC1.0+ relations

As 1NF is easy to oversee in a table, we wont go into detail for this form below.

Telemetry
All attributes are functionally dependent on $\{mId, dId, time\}$

Candidate keys: $\{mId, dId, time\}$

Set of key attributes: $\{mId, dId, time\}$

2NF:

Check for each FD that:

- LHS is a super key or
- RHS are all key attributes

$$\{mId, dId, time\} \rightarrow \{\ ori, pos, vel, sta, link, cpu, mem\}$$

The LHS is not a proper subset of the key, but the RHS is not all key attributes.

3NF:

Candidate keys: $\{mId, dId, time\}$

Set of key attributes: $\{mId, dId, time\}$

Then we check all FDs hold minimum one of following:

- LHS is a super key or
- RHS are all key attributes

$$\{mId, dId, time\} \rightarrow \{\ ori, pos, vel, sta, link, cpu, mem\}$$

LHS is superkey.

BCNF:

Then we check all FDs hold minimum one of following:

- LHS is a superkey

LHS is a superkey
LHS is a actually superkey

Telemetry relation holds BCNF (and hence the others)

DroneRelation

As the drone table only consists of one attribute holds for 1st, 2nd, 3rd and BCNF normal forms

Swam Relation

Candidate keys $\{sId\}$

Set of key attributes: $\{sId\}$

2NF:

Check for each FD that:

- LHS is a super key or
- RHS are all key attributes

Check our functional dependency:

$$\{sId\} \rightarrow \{\ dId\}$$

The LHS is not a proper subset of the key, but the RHS is not all key attributes.

3NF:

Then we check all FDs hold minimum one of following:

- LHS is a super key or
- RHS are all key attributes

$$\{sId\} \rightarrow \{dId\}$$

LHS is superkey.

BCNF:

Then we check all FDs hold minimum one of following:

- LHS is a superkey

LHS is a superkey

Estimate Relation The given time and the drone determines the position and the mission id - they is the determinants. Thee two others are called dependent. Hence this defines the functional dependency.

Candidate keys $\{dId, time\}$

Set of key attributes: $\{dId, times\}$

2NF:

Check for each FD that:

- LHS is a super key or
- RHS are all key attributes

$$dId, time, mId \rightarrow ePos$$

RHS are not all key attributes, as there is none at all.

3NF:

Then we check all FDs hold minimum one of following:

- LHS is a super key or
- RHS are all key attributes

$$dId, time \rightarrow ePos, mId$$

LHS is super key

BCNF:

Then we check all FDs hold minimum one of following:

- LHS is a superkey

LHS is a super key

Route Relation

Without the surrogate key, this relation only consists of one attribute, *route*, hence trivially conforms to all 4 normal-forms.

Mission Relation

The determinant attributes is route id and time_created. The route could be used multiple times, but would be created at the same time. Hence the other attributes is dependent.

Candidate keys $\{routeId, time_created\}$

Set of key attributes: $\{routeId, time_created\}$

2NF:

Check for each non-trivial FD that:

- LHS is a super key or
- RHS are all key attributes

$$time_created, routeId \rightarrow name, done, active, swamId, time_updated$$

RHS are not all key attributes, as there is none at all.

3NF:

Then we check all FDs hold minimum one of following:

- LHS is a super key or
- RHS are all key attributes

$$time_created, routeId \rightarrow name, done, active, swamId, time_updated\mathrm{s}$$

LHS is super key

BCNF:

Then we check all FDs holds following:

- LHS is a superkey

LHS is a super key

Result Relation
The determinant iss the img, the id for the picture coursing the reporting. This is unique

Candidate keys $\{img\}$

Set of key attributes: $\{img\}$

2NF:

Check for each non-trivial FD that:

- LHS is a super key or
- RHS are all key attributes

$$img \rightarrow time_created, loc, fau, dId, missionId$$

RHS are not all key attributes, as there is none at all.

3NF:

Then we check all FDs hold minimum one of following:

- LHS is a super key or
- RHS are all key attributes

$$img \rightarrow time_created, loc, fau, dId, missionId$$

LHS is super key

BCNF:

Then we check all FDs holds following:

- LHS is a superkey

LHS is a super key

Tasks Relation
Multiple drones could start from the same location, in the same mission. Hence, the drone id and the location can determine the rest of the attribute s. But we need to add the mission id to the set of determinants, because the same drone could be at the same location in another mission.

Candidate keys $\{loc, dId, missionId\}$

Set of key attributes: $\{loc, dId, missionId\}$

2NF:

Check for each non-trivial FD that:

- LHS is a super key or
- RHS are all key attributes

$$missionId, dId, loc \rightarrow dspec, iType$$

RHS are not all key attributes, as there is none at all.

3NF:

Then we check all FDs hold minimum one of following:

- LHS is a super key or
- RHS are all key attributes

$$missionId, dId, loc \rightarrow dspec, iType$$

LHS is super key

BCNF:

Then we check all FDs holds following:

- LHS is a superkey

LHS is a super key

What impact would a sorrugate key have had?
If we derived the dependencies with the surrogate key, we would end up with a surrogate key as determants for the natural keys, suggestion a split between the natural keys (determined by surrogate) and the other attributes. This makes not a lot of sense, as we would need to make queries on two tables each time then.

4.5 Lossless Functional Dependencies and Preservation

When decomposing a relation into multiple relations, it is important that we can restore the information by joins. Joins are the only way to recover a decomposition in relation theory. A decomposition is lossless if for any instance r of a given relation R, r is equivalent to the natural join of its projections onto $R_1, R_2, ..., R_n$. Essentially it means that the functional dependencies should be preserved, so the FDs that holds on all relations $R_1, R_2, ..., R_n$ should be equivalent to the original onces. If you have decompositioned a relation, the Chase Test can be used to ensure that there isn't any loss. As we do not have decomposed any relation, this important point would not be showcased. Be aware that it is not always possible to both have dependency-preservation and lossless-joins in BCNF. All relations used here are [22].

4.6 High-Level Database Models to Relations

In practice it can be naturally to move directly from at High-Level Model to the implementation. But as the implementation can be very concrete there is this extra step in theory, wherefrom the actual implementation can be based to different concrete implementations. For completeness it's included. As we are implementing the SQLAlchemy abstractions, this step will result in a definition very close to actual implementation:

Object Definition Language (ODL) is a text-based language for specifying the structure of databases in object-oriented terms [22]. ODL looks essentially like a text representation of UMLs, which make the transformation simple. Another key benefit is it almost a "unified" implementation that makes it even easier to implement ODL in a given database-abstraction.

ODL also uses classes and as UMLs a class can have a name and attributes (and methods, but we dont use this feature). Relationships are embedded in the classes as a part of the properties (contrary from "normal" Relations that has them as a separate class. ODL offers a similar type system as the one used in the C languages (188). Below relationships are denoted as sets, that is has an unordered set of elements, where an element only have one occurrence (in difference from bags), which we have designed the tuples to specifically be.

We have the ODLs for our LiMiC1.0+ databases defined below:

The prototype of ODL:

```
class <name> {
     <list of properties>
}
```

Drone / Logger Relations

Conversion from high-level model to database relations.

```
class Drone (key dId) {
        attribute UUID dId

    // relationship(s)
    relationship Set<Estimate> estimations
    relationship Set<Swam> swams
    relationship Set<Telemetry> telemetries
}

class Swam (key dId) {
        attribute integer rowId
    attribute UUID sId
    attribute string dId
}

class Telemetry (key dId) {
        attribute UUID lId
    attribute Time created_at
    attribute Array ori
    attribute Geometry pos
    attribute float vel
    attribute string sta
    attribute integer link
    attribute integer batt
    attribute integer cpu
    attribute integer mem
    attribute string dId
}

class Estimate (key dId) {
        attribute UUID eId
    attribute UUID mId
    attribute Time created_at
    attribute Geometry ePos
    attribute string dId
}
```

Mission Relations

Conversion from high-level model to database relations.

```
class Route (key rId) {
        attribute UUID rID
        attribute JSON route
}

class Mission (key mId) {
        attribute UUID mId
        attribute string name
        attribute boolean active
        attribute boolean done
        attribute UUID swamId
        attribute UUID routeId
        attribute Time time
```

```
    attribute Time created_at
        attribute Time updated_at

        // Inverse relationship(s)
        relationship Set<Task> tasks
        relationship Set<Result> results
}

class Task (key tId) {
        attribute UUID tId
        attribute string iType
        attribute GNSS loc
        attribute JSON dSpec
        attribute UUID mId
        attribute UUID dID
}

class Result (key rId) {
        attribute UUID rId
        attribute time Time
        attribute GNSS loc
        attribute string fau
        attribute UUID img
        attribute UUID mID
        attribute UUID dID
}
```

This concludes the end of our database design. The trip from ODL, to actual implementation would be very short. Now we would look into the application design before putting it all together.

4.7 Applications Architecture

All applications in LiMiC1.0+ follow the same application architecture concept. When complexity of an application grows it is a huge benefit to organise code according to responsibilities or concerns avoiding spaghetti code [54]. We are using a layered approach, which is a logical separation, as it is simple yet providing many benefits. lower-level functionality can be reused throughout the application, resulting in standardized less code following the DRY principle [30]. This enhance readability, encapsulation and maintainability.
The encapsulation secures that when a change in a layer (or the whole layer is replaced) only the layers that directly interact with this layer is affected. This makes it easier to replace functionality within the application.

4.7.1 N-Layer Architecture

LiMiC1.0+ uses a 3-Layer architecture (see: 4.3), much like the traditional one consisting of user interface (UI), Business Logic (BLL) and data access layer (DAL) [35]. Instead of the user interface layer, LiMiC has a control layer (CL) managing endpoints.
The flaw, with this architecture in difference from other, more complex ones that uses dependency inversion principle, e.g. clean architectures, is that the CL is dependent on BLL which is dependent on DAL. This can make testing harder in general, because it requires a test database. However, we utilise other simple techniques removing the headache and mainly focuses on integration-tests. [36].
BLL-layer is organized in internal services, managing each own area of concerns. This encapsulates the logic and makes is it easier to move an internal service to another microservice of same architecture (that provides the same dependencies), with moving the big components in the service- and controller file (The dictated schema file must also be moved). Further there is some dependencies between these services, that can complicate a migration. This logic is is kept separated to meet the DRY principle. This means that the internal services is not as loosely coupled as the microservices themselves, but is somewhat encapsulated in area of concern, but, with some degree of cohesion.

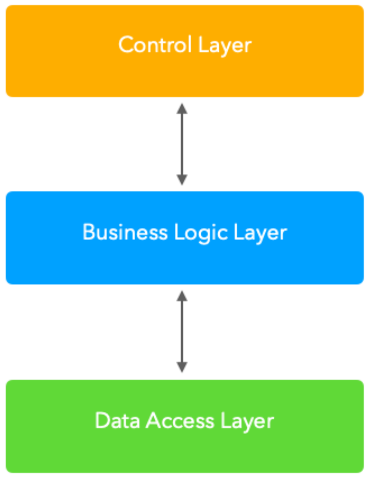


Figure 4.3: 3-Layer Architecture

4.7.2 Applications Logic Devision and Data Flow

Using the layered architecture we need to define each area of responsibility, hence define logically separation within the application. The figure below gives and overview between the architecture and the abstracted areas within the application.

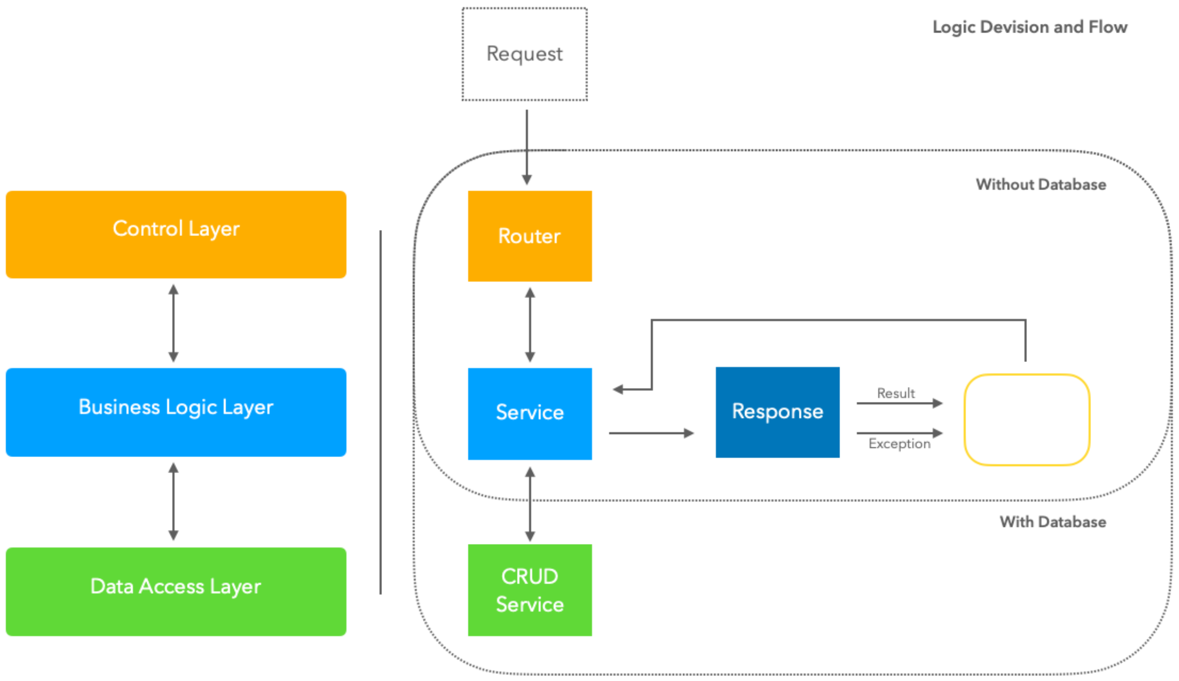
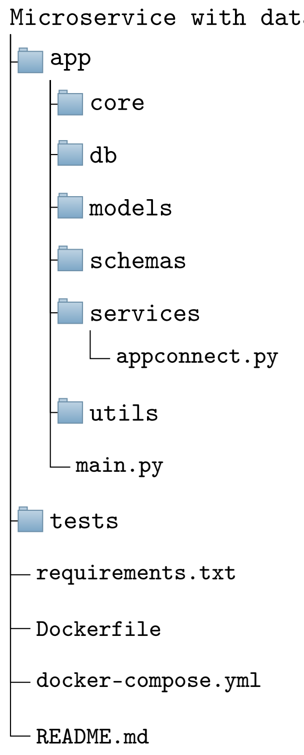


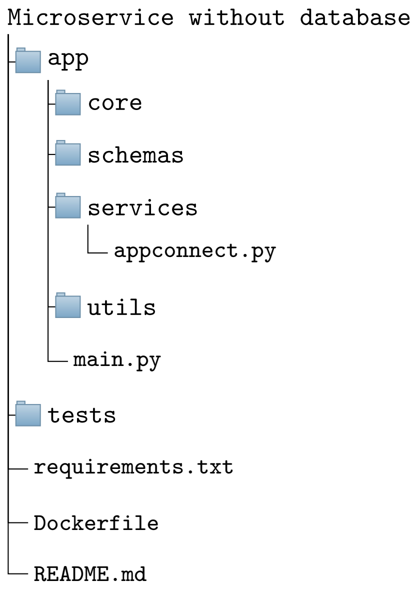
Figure 4.4: Logical Separation Architecture

A request triggers the endpoints in the controller layer, these are handles by the FastAPI Router Objects (see section 4.8.5). An internal service object class is then constructed and a function is called on the instance with given arguments. The services, serving in the business layer can, if the application have a database implemented construct and call the CRUD service in the DAL, communication with databases via the SQLAlchemy Abstractions. The services results returns back to control layer within in a Response object that evaluates and handles either the exception or the value to return.

4.7.3 Base Application Directory Structure

The two base-directory structures of the applications, with and without database. Some applications has extended this structures, this is described in the specific services sections.





4.8 Application Structure and Modules

In this sections the modules that resides in the base structure of the microservices is described. The modules and concepts are used across all applications, unless the parts that is managing the database or the database access. The description is ordered by the module/package and module names corresponding the directory tree.

4.8.1 main.py

Contains the hook-object of the FastAPI application itself (Used by the ASGI Server Uvicorn). Calling the `app` object in the file triggers the construction using the function `construct_application`, which is responsible for constructing the FastAPI object, called app. Custom `routers` is added to this object, one for each internal service. They are all added with a 'tags' arguments for the sake of ordered output in the automatic generated documentation. A python decorator with *pie*-syntax, (in Python this a decorator is a function wrapper, that dynamically can change/conform the function) is used to add a custom exception class and the custom exception handler from the module `core/exception.py`. If the application is implementing the database-abstraction (Mission- and Drone/Logger Service), the `add_event_handler` function, is used to include event handlers [66] for handling the events *startup* and *shutdown*. The actual events are two functions that is triggered in `core/task.py`.

4.8.2 core

The core directory contains the necessary modules for starting and configuring the application from `main.py`.

config.py

`config.py` is also present here and serves as a alternative to a `.env` file. This was a necessity because of the DevOps strategy that relies on .evn files. It is a class that inherits from Pydantic's `BaseSettings` class [57]. This means utilize Pydantic's build in schemes for database URLs, in these services `PostGresDns`. A URL that doesn't fit the

scheme, will get a descriptive Pydantic exception error and hence the application wont even try starting. Further, attributes not declared inside the class but not initialized is tried to be read from the environment.
Besides containing the database URL string, it also contains the URLs to other services that the application need to communicate with.
The values can be called from the multiple functions inside the `config.py` module. Because these values is triggered by the CL-layer and potentially heavily called and do not change after the application is first initialized, there is made use of a Least Recently Used (LRU) Cache Strategy, for a little performance boost. This is especially effective when variables are read from the environment (injected in the Docker files) or using files as `.env`. Python's standard implemented `functools`, makes it a very easy implementation with the *pie*-syntax decoration [58]. We do not make use of any of the advanced settings, because or cached values are not shifting heavily, but only a "read-in-once" feature from the resources.

tasks.py

If the application uses a database this module is present.
We define the two functions, that each returns a asynchronous function. `create_start_app_handler`, is responsible for establishing a connection and for calling the function `create_tables`, to create all database-tables (only created if they don't already exist). `create_stop_app_handler`, is responsible disconnection. They actual connection is done in `db/tasks.py`.

exception.py

To be able to give more meaningful response error messages from the business-layer, a custom exception class is implemented. This is done in the class `ExceptionCase`, which implements the constructor `__init__` and the `__str__` for representing the class object as a readable string [52]. The attributes of the class is `exception_case`, `status_code` and `context`. exception_case is the class-name of `self` object. This would be the name of the initializing sub-class. Context which is a dictionary, that enables very flexible information-passing. These attributes is overwritten by the sub-classes inheriting the ExceptionCase class. The sub-classes are used to define the names for the exceptions, see figure 4.5.
As explained in section main.py, the standard *HTTPException* handler needs to be overwritten in the FastAPI application [69]. We overwrite it with `app_exception_handler`, which returns a JSONResponse (FastAPI response type defined in `fastapi.responses`), with the HTTP status code and a context from the exception.
The class `AppException` is containing the class-definitions of the actual custom exception objects. These sub-classes is not more of a constructor that is initializing its superclass ExceptionCase. The actual integer-values is typed directly in the status codes. The readable variables from `fastapi.status` is used. The context dictionary can be passed as argument during creating. This means values or other "in place" information can be parsed. This is heavily used in the internal services.

```python
class Create(ExceptionCase):
    def __init__(self, context: dict = None):
        """
        Creation failed
        """
        status_code = status.HTTP_500_INTERNAL_SERVER_ERROR
        ExceptionCase.__init__(self, status_code, context)
```

Figure 4.5: Creation of custom create exception

This implementation secures a very readable and easy understandable exception handling in the business logic without knowing these implementation details, as seen in the figure: 4.6. See an example on what the client receives in appendix: 9.9.

response.py

Contains a class called `Response` which supports a generic outcome, allowing the returning of both exceptions and actual results encapsulated in the Response attribute `res`. Response's constructor takes one argument, `res`, that can be an actual return value or an exception. By testing the the res object is an instance ExceptionCase, we can

```
def create(self, _in: TelemetryBase) -> Response:
    tele = TelemetryCRUD(self.db).create(_in)
    if not tele:
        return Response(AppException.Create({"with": _in.dict()}))
    return Response(tele)
```

Figure 4.6: Creation of custom create exception

determine if res contains a exception or not. If the object is an instance of the exception case the three of the four attributes is set accordingly. This means `success` is set to false, `exception_case` is set the exception_case, that must be in `res` and likewise with the `status_code`. On the other hand, if it not an instance of ExceptionCase, we set the `success` to true and the two other to `None`. In either case the fourth attribute `value` is set to `res`.
Response implements the magic methods `__str__`, `__enter__` and `__exit__`. The first simply returns a user-friendly string-representation of the object. The two latter is to serve the use of a Context Manger in the handling Response objects (see 4.7). `__enter__` defines the entering for the runtime context related to the object. `__exit__` is implemented to conform to the context manager. It is used to exit a runtime related to a object and describing the exception that course the context to be exited [51]. We do not use this functionality, as we either return a value or exactly raise an exception inside.

Inside the response.py module the function `response_handler` is defined, which responsibility is to deal with the Response instances unpack the given value in `Response().res` and return it. This is done with a *With Statement Context Manager*. The `with` statement will bind the objects return value (it extracts using `__enter__`) to the target specified in the `as` clause. This allows us to implement a very readable response handler, even with a generic object, as seen below.

```
def response_handler(response: Response) -> Any:
        """Handling Response"""
        if not response.success:
            with response as exception:
                raise exception
        with response as response:
                return response
```

Figure 4.7: Response Handler for internal service response

4.8.3 db - Database

Only present in applications implementing database abstraction.

database.py

This module is responsible for constructing the database *engine*. Using the `get_config_database()` the modules calls the database url form the `core/config.py` file. The engine is the starting point for using SQLAlchemy. It's the base for Python Database API Specification (DBAPI), that is delivered to SQLAlchemy application through a connection `pool` and a `dialect`. The dialect makes the abstraction between multiple database systems possible. A DBAPI, or in a more general lingo, driver is psycopg2 [90].
The function `create_engine` creates a Dialect object tailored, in our cases, towards the PostgreSQL and a Pool object that establishes a DBAPI connection at the database url and port. This happens when the first connection request is made, which is not made during this lazy initialization [86].
It is possible to use the engine for directly interact to the database, but here, it is use as a argument in a `sessionmaker`, to create a *(*Session) object, that enables object relational mapping (ORM).
When using ORM, we start by configuration the database tables and by defining the classes which will be mapped to those tables. In SQLAlchemy this functionality us done together using the systems of `Declarative Extensions` [78]. This means we can creates classes that include directive to describe the database table they will be mapped to. Mapped classes are defined using a base class (declarative base class), that maintains a collection of classes of

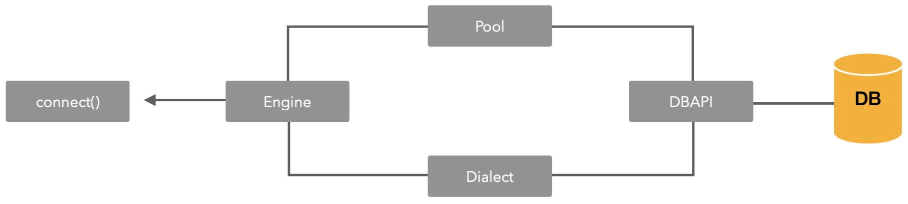


Figure 4.8: SQLAlchemy Engine, Pool, Dialect and Driver

tables to that base. Applications usually only have one `base`, so do the once in LiMiC1.0+. It is declared using the `declarative_base()` in the file and imported into the defining classes (`models/`). This are the Models in our application, that all implements Base to implement the functionality just described.

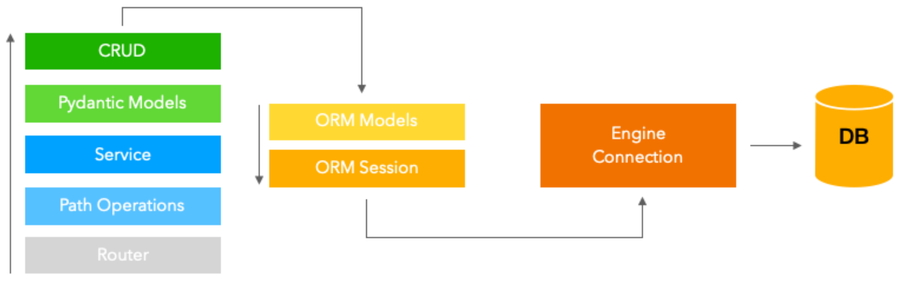


Figure 4.9: SQLAlchemy Database Interaction

sessionconnect.py

A independent database session (hence connection) per API request is needed. Using the same session on the whole duration of the requests is running, and then close/release it after the request it done. This is done in the `get_DB` function by creating a new session object as dependency with `yield` in a `try`, `finally` setting. The injecting this yielded value into the path operations (in routers) and the 'finally' key word make sure that after the response have been delivers that the session is closed [65]. The injection and constructions make sure that the session is always closed after the request, also if there was an exception during the request.

tasks.py

Responsible for the connection between FastAPI and PostgresSQL, together with table-related database tasks.
We are using the async interface `databases` that supports PostgreSQL, MySQL and SQLite, to create the database connection using the `Database` object. The applications do not rely on async implementation, but know it is a open possibility for future extensions (see why in section 4.8.8). We initializing a connection-pool of size 5-20, as this is the standard used in the documentation [12]. The connection are established and torn down in a secure block with `try/expect` with the use of the `await` keyword inside. After the connection is successfully established, it is attached to the FastAPI state object (app context) using the key: `_db`. When shutting down we use this key to get the object and call disconnect() on it. If an exception is triggered during any of the events, it is outputted to the consol/terminal using the Python logger library, for user friendly formatting.
This module also implements functions for creating and dropping database tables. This is done using the `Base` (declarative_base) object, which holds the table-names in its meta-data object. We are customizing the construct compilation SqlAlchemy's `DropTable`. This is done using a decorator `@compiles` (from `sqlalchemy.ext.compiler`), which changes the internal DropTable function on the Postgres dialect (other dialects is not affected), to adding "CASCADE" to any drop table statement [81]. This is only used in the staging environment, to force dropping tables, even if they contain foreign keys in other tables, with that assumption, that all tables is being dropped, hence dangling primary- and foreign keys resolved.

4.8.4 Models

Only present in applications using a databases.
Models are the mapped classes defined by inheriting the `Base` class (declarative base class). Base is imported from `db/databases.py`. There is one model for each relation in the database. These models are used directly by SQLAlchemy to create the actual database schemas inside the database. Every attribute is `Column`-class, constructed with the given attribute type, from SQLAlchemy. All types of variables are imported directly from SQLAlchemy unless the `Geometry` objects. A typical declaration of such a class is shown in figure 4.10. The `as_uuid=true` in the `UUID` Column, configuring that the attribute can be parsed a whole UUID object.
Relationships [82] is very easy to define (see figure 4.10), mission id attribute is defined with a foregin key attribute. This means it is child or "many" in a one-to-many relation. The corresponding line in the missions model is: `tasks = relationship('Task', backref='mission', cascade="all, delete, delete-orphan")`. This line defines the relationship between a Mission and its tasks. The backref makes is possible to load a tasks-object and call the mission on it, receiving it (this is ORM functionality). The cascade definition [83], defines when we are deleting a mission, all the orphans, the tasks, would be deleted also. This is the only actively ORM-object functionality, used in the deletion of Missions.

Geospatial database - using Geoalchemy2

Geometry types is imported from the Geoalchemy2, that supports all geospatial types that PostGIS supports. Creating a geospatial type is done using the `Geometry` object, passing in the geometry type in a string and the number of dimensions, which is here 3 as we use a point with a z-value also.

```
class Task(Base):
    __tablename__ = 'tasks'
    tId = Column(UUID(as_uuid=True), primary_key=True)                  # id
    iType = Column(String)                                              # inspection type
    loc = Column(Geometry('POINTZ', dimension=3))                       # location
    dSpec = Column(String)                                              # data specification
    dId = Column(String)                                                # drone id
    # Relationships
    missionId = Column(UUID(as_uuid=True), ForeignKey('missions.mId'))  # foregin key
```

Figure 4.10: Declaration of Mapping Class inheriting declarative base

4.8.5 Routers

Routers are the core of the control layer and exposes the API's endpoints to the outside world. A router is constructed from FastAPI's `APIRouter` that is then included in the construction of the application (4.8.1). Each router is dedicated for the area of concern, for one internal service. Because of the desire of a clear separation in the API's URL's, every instance of Router is given a prefix as argument on initialization, which it applies to all URLs. In LiMiC1.0+ the implementation of the routers (hence the control) is very similar and trivial thanks to the underlying abstrations. As seen in figure 4.11, a router endpoint is created by a decoration `@router` decoration, calling given `REST` function, with the `path` as first argument argument. In Fig. 4.11, the second argument is `response_model`, which is always a Pydantic class or a string value in this implementation. `.delete`-endpoints have a status_code instead a response model, as the outcome it not a response, but a status on the instructed action (other operations has status codes encoded in the response as described in section 4.8.2). The function name inside the decoration is important, as it it used by the automated documentation. The function's arguments, that is set to an default value or a simple type, is should be treated as URL parameter values. On the other hand, if it is a class (here pydantic class), it is treated as a body of the request. This is exactly here we use the validating functionalities of Pydantic. If the endpoint use a database, the database-connection is injected here using FastAPI's *Dependency Injection* System [64]. As explained in section 4.8.3, we want a database connection for each request, this is managed by exactly this injection-logic, using `FastAPI.Depends` to request a connection from the function `get_DB` in `sessionconnect.py`.
The body of functions is always constructing an instance of the corresponding internal service (with the database connection as argument, if used). Likewise, the corresponding function in the service is called with the potential

```python
@router.post("/new", response_model=DroneOnCreate)
def create_drone(dId: str, db: Session = Depends(get_DB)):
    """
    Create Drone
    """
    response = DroneService(db).create(dId)
    return response_handler(response)
```

Figure 4.11: Declaration of Mapping Class inheriting declarative base

arguments. If the endpoint is not a delete-request, the service will have a response, which always is a instance of `Response` and handles by the `response_handler` described in 4.8.2.

4.8.6 Schemas

Schemas is the Pydantic [56] classes used for validating and parsing heavily in all the three applications. Pydantic is very powerful yet flexible and offers a lot of advanced functionality using a Pythonic philosophy. The specific functionality is described is described with the use cases in the service descriptions.
All SQLAlchemy models have one or more Pydantic classes. Pydantic offers mapping between the SQLAlchemy Models and pydantic classes when the `orm_mode=True` in the `Config` subclass of the pydantic classes. This is enabled for support, but not primarily used as we do not map directly because of the conversion of the geospatial attributes. All super classes inherits Pydantic's `BaseModel`. Attributes can be defined as simple types, or other pydantic classes.
The use of these schemas allows the application to validate the input already at control-layer level before parsing it to business-logic. Likewise the exported response_models is also automatically validated before returned. Exactly this forces developers to take care of the excepted data-shapes around the distributed system, as the application would fail internally, if the schema is not updated according to what is returned in the internal logic. Hence each service is validating its input, but do also have internal self-justice of schema-changes. This leads us to another nice benefit. Serializing the response using Pydantic in one microservice allows you to use the exact same class to map/deserialize it into a class instance from `JSON` in the receiving service.
Schemas is used directly from CL directly through BLL and into DAL where it is mapped to SQLAlchemy models. Outcomes from DAL is also constructed as schemas. It is possible to return SQLAlchemy models directly, but then FastAPI tries in 'ad hoc' manner to fit the models and there is no guarantee, therefore for consistency this is done all over.

4.8.7 Services

Internal services within each microservice. These services are in general encapsulated but do to some degree depend on each other. This is mainly static methods, that is kept in its concerning area for consistency and readability. The internal services share the same application core, but have separated files in CL, BLL and DAL. This makes an easy immigration to other services having a similar application core (cross-service static functionality need to be copied into migrating service, likewise with given Pydantic Schemas). The structure can be seen in fig. 4.12.

Databases implementing database abstraction

Internal services relying a database connection the must implementing `SessionConnect` class residing in `services/serviceconnect.py`. `SessionConnect` only have a database session (named `db`) as class-attribute. The two classes `AppService` and `CRUD` is inheriting `SessionConnect`. This does not any new functionality, but enhance the readability, when they are inherited by the internal services classes or the corresponding CRUD classes. Further it allows separate shared attributes or functionality in later. This implementation abstract the database Session object totally out of the service classes, but it is callable in the `self` object. The relation can be seen in fig. 4.13.

4.8.8 Asynchronous application

Asynchronous execution is an important aspect for microservice architecture. Modern Python offers a very intuitive and easy definition of asynchronous code. This is done using the `await` keyword inside a function that is defined

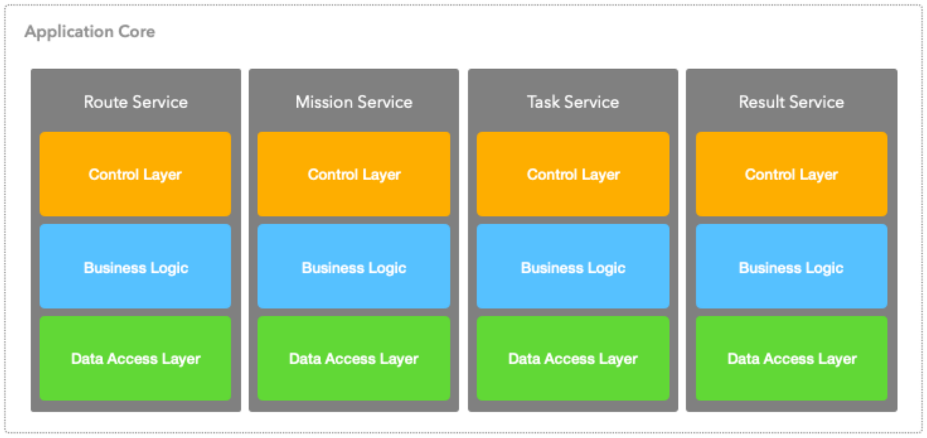


Figure 4.12: Mission Service - internal services

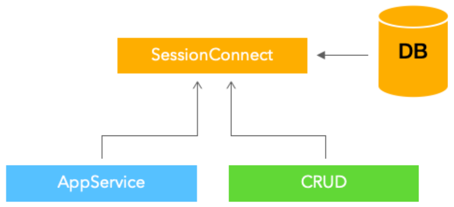


Figure 4.13: Session inheritance in BLL and DAL

with `async def` (the two standard keywords to defined coroutines in Python). FastAPI is based on Starlette [11] which are based on AnyIO, making async implementation using the both libraries compatible with `asyncio` and `Trio` [63]. It is important to realize that we are considering Python implementations. Python runs in a single thread which results in some complications with the conceptual theoretical ideas, because of the Global Interpreter Lock (GIL). Async IO in Pythons concurrency libraries is single-threaded, in a single process. This is called cooperative multitasking. This means that async IO give the feeling of concurrency, but as the coroutine (name in asyncio) can be scheduled concurrently, they are not in fact concurrent executed. All in all, asynchronous is closer to threading than multiprocessing and is not at all parallelism. Asynchronous means that computations (coroutines) can be paused while waiting for some dependency (e.g. I/O) allowing other coroutines run on the "single tape machine" in the meantime [3].

FastAPI is very pro-async, but can still implement non-async libraries. This requires that endpoints in the CL is defined properly. By doing this, the performance benefits for the `async` event loop can still be utilized - as long as the non-async libraries only block on IO. This is possible because a path operation function declared with "normal" `def`, is being ran in an external threadpool [63], that is awaited, instead of a direct call, which would block the server thread. Blocking I/O called in a path operation with `async def`, the server thread will remain blocked while waiting for response [5].

Asynchronous database communication

As SQLAlchemy now supports asynchronous behavior this is investigated.
A asynchronous database communication have a lot of potential wheen writing `HTTP` like chat servers, that specifically need to concurrently maintain large numbers of arbitrarily slow or idle TCP (established connection) connections (Arbitrary round trip times (RTT) but a maintained throughput). The strategies on asyncio is never necessary and will likely directly hinder performance in applications with standard CRUD oriented database code [37]. So why is this? The answer should be found in the understanding I/O - and CPU bound execution. The main part of the execution-time is not I/O bound in a Python application with database dependency, taking into account (Our CRUD-application and database is not distributed). Even taken a network connection into account (unless it is very slow and would in fact bring up I/O bounds) [37]. This is contrary from expectations in compiled languages

as C, or maybe even Java, that uses Just-In-time compilation (JIT) (interpreted) from compiled bytecode to native instructions. Python is no nearly as fast as the databases in general, yet there could be runtime exceptions, when using large OLAP-queries. The PostgreSQL database server is implemented in C [50] (actually uses JIT compiling of its expressions [49]) and is significantly faster than interpreted Python. Further we are using connection pooling that removing much of connection overhead. Pythons own overhead in constructing query-objects, handle the results keeps the CPU busy, which really shadows the throughput advantages to the database that non-blocking IO would provide.

Even if we find all the race conditions in the process of using threaded code or asyncio's coroutines, it does not really matter when communicating with RDBMS [37]. This is really important point when deploying in distributed horizontal ways. When implementing database code, there is exactly one strategy that will assure correct concurrency this is's by using ACID-oriented constructs and techniques. This is a complex problem, that we, in this implementation use SQLAlchemy to help implementing in a correct way [37]. The underlying PostgreSQL database used in this implementation, is also ACID-compliant [49]. If we really need non-blocking code, we should move the connection-pool logic to the application itself, having the BL/DAL-CRUD code behind the pool [37], breaking the idea with FastAPI. Many concurrently request to application logic could benefit from the asynchronous implementation, but most of them are bound to a DB, hence this stack probably introduce quite a limit to vertical scalability.

To sum up. FastAPI implements very powerful asynchronous features. Even synchronous tasks runs in a async manner thanks to the external threadpool. Likewise in-between microservice communication can serves from these benefits also, e.g. using FastAPI's backgroundtasks feature. The a async database connection is not a great idea in this type of application.

4.8.9 Uvicorn

FastAPI runs on top af a web server, Uvicorn [13], and it is partly thanks to this, that FastAPI is fast. It it trivial to use webservers, when you have a library like FastAPI implemeting the ASGI standard, but there is a few things to consider anyways. Therefore we address these concerns. Uvicorn is a very fast Asynchronous Server Gateway Interface (ASGI) server, implementing *uvloop* [13]. UVloop is a fast, drop-in replacement for the asyncio library's event loop. UVloop is written in *Cython* hence implementing functions and types from the C language [87]. It makes asyncio fast, actually at least 2 times faster than nodejs, gevent etc., close to the performance of *Go* programs. UVLoop makes it possible to implement networking code in Python, that can push 10k+ requests per second per CPU core, meaning it can be even further scaled on multicore systems [92]. Be aware that Uvicorn is relying on replicating worker processes to make this scale [71]. LiMiC1.0 is deploying in clusters using Kubernetes on multiple physical machines (potentially). In this case its more fitting to control the replication at cluster level instead of using the process manager in the single container. Kubernetes have built in functionality for handling replication of the containers while still supporting load balancing [72].

One Load Balancer in the distributed system

One of the benefits by using a distributed manager system like Kubernetes is the load balancing between replicated containers. This is handles by the internal networking mechanism, that listens on the main port and distributes the communication to the potential several containers running a replicated instance (this is done by distributing the requests in turn). As explained in 4.8.9, we want to want to control the replication in cluster level, so normally these identical containers would in our case run a single Uvicorn process running our FastAPI Application. This creates basis for utilizing parallelization over different cores or even physical machines [72].

4.8.10 Dockerfile

Each application also contains a Dockerfile for containerization. See more in section 5.

4.8.11 Docker Compose

If the application is using, hence dependent on, a database it is containerized using a multi-container system. See more in section 5.

4.9 Significant Dependencies

4.9.1 Python

We use python as a programming language in this implementation. Even though the benefit of distributed systems allows us to switch to use-case specific solutions, there is not any reason to change away from the base ideas and argumentation introduced in the LiMiC 1.0 implementaion.

4.9.2 FastAPI [70]

As explained in the LiMiC 1.0 section FastAPI is very performant an comes with great functionality and benefits that would as use full in the product of the next services.

4.9.3 Pydantic [56]

Pydantic was very shortly introduced latter sections. The official use case of Pydantic is data validation, but the framework is very flexible as powerful the use cases are many. In this project Pydantic is both used as parsing-validator serving below FastAPI, used as main-engine in a parser and used for parsing application configurations.

4.9.4 SQLAlchemy [80]

Is a Python SQL toolkit and Object Relational Mapper. SQLAlchemy provides efficiency high-perfromant database access and serves as an abstraction between database sepcific communication and the Pythonic domain language. This abstractions opens for the possibility of changing the underlying database, but the code base remain the same. Furhter SQLAlchemy handle, drivers, connection-pools etc. SQLAlchemy translates the Query objects we defines into SQL-statements under the hood. Hence we do not need to handle any SQL at all.

4.9.5 GeoAlchemy2 [16]

GeoAlchemy 2 provides geospatial extension to SQLAlchemy, making it possible to handle geospatial databases. This is also an abstraction, but the focus is on implementing support for PostGIS. This means that it is very easy to move functionality across application layer and into the database. Some of the functionality in GeoAlchemy2 builds directly upon Shapely

The following sections is describing the specific microservices and will refer to the general concepts described in section 4.7 and section 4.8. Each service will be described by its internal services in sections of each REST segment, addons tools to the service directory and tools in the utilize folder.
Mainly the Business Logic Layer (BLL) and the Data Access Layer (DAL) would be described with reference to the CL Layer.

4.10 Missions Service

Mission services, can be found in the directory `missions` (in the branch) and consists of four internal services shown in Fig: 4.12. Missions service is integration with the VRP and the Drone Service.

4.10.1 Route Service

By requirement it is possible to request a new route trough the Mission Service. The Route with its specification is encoded in the initial request to the service. The request is responded immediately, with the `rID` for querying the route. The tasks is executed in a FastAPI backgroundtask [61].
`RouteService()` is a class that resides in the Business Logic Layer and implements the class `AppService`. This means the class takes a database `Session` (connection) as argument when constructed. Hence the database connection can be called using: `self.db`. The connection is often injected in the CL layer path operation using dependency injection.

Mapping Class

The Routes Mapping class can be found in `models` directory (4). The Route is inheriting the declarative Base class, as earlier described. The class itself is declared with an `rId` as the primary key using, using the UUID constructor with *as_ uuid=True*, which allows UUID instances to be parsed directly. The other attribute is the `route` of type `JSONB`, a custom PostgreSQL-type. Both attributes is declared in `Columns` objects used to SQLAlchemy settings.

Router - Control Layer

The concepts of the router modules is very similar. Only specific or unique changes are described here. The router for this internal service can be found in `routers/router_router.py`. The function inside the decoration for *POST*, is quite different and have `BackgroundTasks` as parameter (note that BackgroundTask, can't be used in path operation). When the path operation has returned, the background tasks added to the object is executed. By adding the parameter in path operation in CL enables a lot easier and FastAPI handled implementation.
The *DELETE* operation do not have a return value, but returns a `204` no content return code as success.

Create

Using the `RouteService()` object the function `routeTaskManager` manager is called with the two arguments:

- The body from the requests, with type of RouteRequest (Parsed and validated to an object thanks to Pydantic)
- The `backgrounds_tasks` object, to add a tasks

`routeTaskManager():`
The route tasks manager is generating the id, that the route later can be queried with (becomes primary key). This is done using the built in `uuid4()`. `add_task` is used to add a tasks to the backgroundtasks object. This is done by parsing the function-name itself as first argument followed by the functions arguments as the next arguments. We want to run the function `routeManagerBackground` as background task with the id and the RouteRequest object as arguments.
Last the schema RouteOnCreate2 is used to create a response containing the id. This is encapsulated, as described earlier, in a `Response`.

`routeManagerBackground():`
The RouterManagerBackground is responsible for orchestrating the request of the route from the dedicated static function `get_VRP` and saving it using the likewise dedicated function `save_route`. `get_VRP` with the URL from core/config (using the function `get_config_VRP`) and the requested-body, directly parsed from input.
`save_route` is called with the JSON response got from `get_VRP` and the parsed id. SQLAlchemy returns a `None` type, if there is any problems in DAL. Therefore we check if the responses contains exactly that. If that is the case a custom AppException of `Create` instance is returned with the relevant information.

`get_VRP():`
Is responsible for calling the external VRP Service. Using a safe block (try except in Python), the functions construct a post request using the build in request module. The post request, consist of a body (JSON) containing the routing request send to the passed URL. If the request somehow fails, the request_handler is raising a custom Get exception, containing the VRP's error response. On the other hand, if the request is successful, the request is returned to `routeManagerBackground` in JSON format.

`save_route():`
`routeManagerBackground` calls this function to save the raw JSON into the database. By creating a `RouteCRUD` instance with a database Session as argument, and calling the create_local function (DAL) with the JSON and the id (that the user already have received at this point).
To make a insert operation, a instance of the mappable Route (model) is constructed. Then the object is added to the session using `db.add()` and committed using `db.commit()`
We want to tell the BLL how it went. If there is any problems during inserting SQLAlchemy would throw an exception that would be shown internally, but exposed to the user, in this case not even error 500, as we are running this is in a background task. This is also why it doesn't make sense to create custom exceptions for the users in this case. An arising question here, could be why we don't use this time to create Task objects right away. The first reason is the JSON format is used directly in the UI to display the computed route. This way it is possible to

request the route multiple times before making it a Mission.

The second, more important reason is object association. Having Tasks that maybe is associated with a Mission, would require a link/association table to querying for specific tasks associated to a Mission. This would require two DB requests each time. By postponing this creating, we can associate a Task with exactly one Mission.

Pydantic Schemas

The schemas uses for route can be found din `schemas/route.py`
Most of the schemas is self explanatory containing the `UUID` and the response in a string (Pydantic supports JSON as type, but have some weird behavior).

RouteRequest: The schema RouteRequest, handling the initial request from the UI. the schema (class) is a expected argument in the path operation. The class itself only have one attribute, named body: `dict[str, list]`, that matches the body of the request. Using the `@validator` decoration, a custom validation is defined. The `pre` keyword causes the validator to run before all other validation (Is anything is added in the future, this essential validation running first). `check_dict_structure(cls, value)` runs the validation. Arguments is the cls, a class object that can access the state of the class-object and the value, referencing the 'body' variable defined.

The values of the keys is validated. They can only be either 'sources' or 'targets'. Next the lists is iterated and all values is validated. If the keys is sources, the list contains a lists, each containing a drone id, tower id, latitude and longitude, hence length four. If the key is targets, the list contains lists, each of tower id, latitude and longitude, hence length three. The types is checked on the relevant values, likewise the coordinates values is validated to be within ranges. These checks are necessary to ensure that the VRP-service does not return an error. A invalid request would already here be denied, automatically returns Pydantic's error message, and we are not wasting request through the distributed system.

Read

The route relations entities can be extracted using their primary key (PK) - the id, that was returned back to the user. The constructed RouteService instance is constructing a instance of RouteCRUD and calling the function: `.get_by_rId(rId)` (rID is th PK), a DAL function. Then we are using the simple query: `db_route = self.db.query(Route).filter(Route.rId == rId).first()`, to make a query with the Session, on the Mapped class Route, filtering for tuple containing the given rId. We are instruction to stop the search when finding the first occurrence, as we know the PK is unique.
The response is mapped by SQLAlchemy to a Route Class instance, where from we create a instance of the Pydantic class `RouteInDB` is constructed and returned. This follows our convention, that everything leaves DAL in a Pydantic Class (hence validated). This class consists of the rId, with the JSON encoded route. In BLL the response is checked for None-value. If it is a appropriate `AppException.Get()` exception is returned and raised. In the contrary case, the response is simply returned inside a Response object. Because it is a Pydantic Class FastAPI can make it to stringified JSON, that is can return.

Update

A route cannot be updated. As it is possible in PostgreSQL to make queries inside JSON and change the key/values, it does not make much sense in this use case. It is simpler and easier just making a whole new request.

Delete

RouteService have the function `delete_by_id` taking the given `id` as parameter. The function have the very broadly used structure, where a RouteCRUD object (DAL) is created with the Session as construction argument, and the given CRUD-function called on thee constructed object, here `delete_by_id`, and the response from DAL is checked to be 1 or above. A exception is returned if not. otherwise nothing is returned. Any issues with SQLAlchemy would create internal errors and the error is written out in the terminal (all handled by library). A issue triggering in BLL, would be if there isn't any Route with this id. This would make SQLAlchemy None (this is not a DB error), which is captured and coursing the construction and returning of `Delete` exception with 404 error, together with the relevant description in the exception.
The DAL function with the same name is called with the id as argument. The function can be seen below:

```
    def delete_by_id(self, Id: UUID) -> int:
        """ Delete a result by ID """
        num_del = self.db.query(Route).filter(Route.rId == Id).delete(synchronize_session=False)
        self.db.commit()

        return num_del
```

Figure 4.14: Data Access Layer - Delete Route by id

Thanks to SQLAlchemy's Query API [84], the queries structures look very much like each other for the most operations. The DB session is used to call the Route relation, filtering by the id and delete all rows matching (only one, as we are querying for a unique attribute). The `.delete` function is called with the `synchronize_session=False` as parameter, which controls the synchronization strategy to delete (or update) the objects in the session. This setting have to do with ORM (Object Relational Mapping), that we uses with SQLAlchemy. In reality everything can be staged like a series of objects that is maintained by the Session object in memory. It is only reflected in the databases when a `.flush()`, `.commit()` or right before a new query is performed. 'False' means we do not synchronize the session. This option is the mosts efficient and reliable when the session is expired (In this implementation after a `.commit()`). This introduces the bad side effect, that objects, that is updated or deleted in the database may be in the session with invalid values [79]. This does not effect this implementation as we only uses this feature for deletion. The reader will be discover that ORM related functionality is only used when fitting, and not as the main concept. As it makes the concept of working with RDBMS more simple for beginners, it introduces complex side effects and can be harder to grasp for people who knows the RDBMS concepts before hand.

4.10.2 Mission Service

The internal missions service is responsible for creating Missions and tasks from the VRP-response saved in the Route table. Further a missions status is controlled via this service.
`MissionService()` is a class that resides in BLL and implements `AppService`. Refer section 4.10.1, to this means.

Mapping Class

Missions Mapping class can be found in the `models` directory. (see appendix: 9.3).

Router - Control Layer

The missions router, found in `routers/router_mission.py` does not introduce anything new. Refer to section 4.8.5.

Pydantic Schemas

The schemas can be found in `schemas/mission.py`. They do not introduce anything not already explained. This the Pydantic classes for Mission corresponds closely parts or the whole Mapping class sets of attributes, depending on when they are used.

geometries.py
Geospatial positions is used throughout the all internal services in Mission Service. In `core/schemas/utils/geometries.py` the two Pydantic models Point and Position is declared. A Point consists of a Position (can later be extended to e.g. a Polygon consists of multiple positions). The Positions values is validated during construction using Pydantics `confloat`.

Create

A Mission creation is orchestrated from the `MissionService()`'s (constructed in CL) `create` function, taking a schema MissionBase as parameter, containing the missions name and the route id.
A `MissionCRUD` object (DAL) is constructed and the object's `create` function is called, taking the MissionBase as parameter. In this create-function, a swam id and a mission id is generated (as explained, we do not need to consult DB during this). Together with the values from MissionBase we Mission class is constructed (other values

set default by DB). This object is added and committed to the database. A tuple with the two id's is returned (Here breaking with the idea of only Pydantic classes out of DAL).
The route is queried from DB using `RouteCRUD` with the route id.
The tasks and drones is then extracted using `MissionCRUD`'s static function (no object creation needed) named `prepare_route_data` with the route and mission id as arguments. Tasks are committed to the DB using the function `create_tasks`. A Swam is constructed using a call to Drone/Logger Service done by `create_swam`.
Last the mission id is returned.

`prepare_route_data()`:
Take the raw encoded data in JSON extracts and construct TaskBase objects and collects the drone ids.
The JSON data is parsed to traversable dictionary using `VRP_Parser`. Each key in the dictionary corresponds to a drone id (VRP Service convention). Each key is appended to a drone array, before using the key to extracts each drones route, which is a list of locations (in this version only the latitude and longitude is extracted). A drone can have a empty list, hence no route. The drone is still added, to the mission, for now - so it would behave like expected on user side (adding the drones that is picked). For each location in the lists there is created a TaskInDB Pydantic class instance. By doing this, we are validating each value as a part of the process. Each service can't rely on other services integrity in a distributed system. That would make more tightly coupled services, where updates in one service effect another. `iType` and `dSpec` attributes it hardcoded for know, as there is currently no logic/functionality to decide those. The location (`loc`) is a Point class constructed using the Position class 4.10.2. Here a artifical height is also injected to bridge the data-gap, that VRP does not consider drone height at all. It is set to 10 (meters) as standard, for know, because much weather data is measured in this height. This makes is possible to call the drone height meaningfully, from other services already. The mission id is serving as foreign key (FK). All tasks are added to a array, which is returned in a tuple with the drone array

`create_tasks()`
Calling the `TaskService`'s `create_all` function().

`create_swam()`
Orchestrating the creation of a swam, called with the drone service's URL from the config file, the drones and swam id. Creating a expected type of SwamBase, containing the swam id and the drones (a list of strings). The function `construct_request_body` JSONifying a request body with FastAPI's `jsonable_encoder` [89]. `call_drone_service` is calling the drone service endpoint responsible for creating the Swam. This is done in a `try except` safe block. If An exception is thrown `False` is returned coursing a exception being raised of type Create, back in the BLL create function.

`VRP_Parser()`
The `VRP_Parser` can be found in can be found `Parsers/vrp_parser.py` directory, and is the a one of four classes in the module, that is not Pydantic class. `VRP_Parser` is initialized with the VRP, JSON response. The class have one function called `.parse()` Using the Pydantic Keyword `__root__`, a custom root type [55] is constructed in the class `_VRP_response`: `Dict[str, _Path]` (drone id, path). `_Path` is containing the distance and path, which is a list of `_pathItems` containing all the data for each location, including the coordinate. All the values is validated using during this mapping of the classes over the JSON string constructing the dictionary used in `prepare_route_data`.

Read

`get_all()`
`MissionService.get_all(skip, limit)` constructed an called in CL. Skip is the number of entities in table that should be skipped and the limit, is the maximum number of entities returned in one query. This function only returns the 'Metadata' of the missions. This is mainly for performance reasons. The function call the `get_all` on a `MissionCRUD` object constructed. Get all the Query API to make a query on the Missions table with the functions `offset()` and `limit()`. The last `all()` function makes SqlAlchemy returns the result as a list. During iteration of the list, there is created a `MissionInDB` instance for each object and appended to a list, which is returned to BLL function, where is is returned in a Response object

`get_by_mId()`
Called on the `MissionService` object constructed in CL. The function takes a mission id as parameter. The BLL functions is almost identical to the one described above (and many others), with the exception of the function called

in DAL.
The DAL function in `MissionCRUD`, with the same name is called. The function is execution the query seen below:

```
db_mission = (
        self.db.query(Mission)
        .options(joinedload(Mission.tasks))
        .options(joinedload(Mission.results))
        .filter(Mission.mId == mId_in)
        .first() #.one_or_none()
    )
```

Figure 4.15: MissionCRUD().get_by_mId() query

The query structure is similar to the others used, but introduces the use of `joinedload`, which joins (default left join) the relations defined between a Mission and its tasks and results. There several Relationship loading techniques, where is an eager one [85], loading them directly with the first query. Another possibility is to utilize ORM and then call e.g. tasks on the object, that would make SQLAlchemy make another query on demand. When is endpoints is called we also want all data and the readability and it is more obvious how many queries that is performed.
It the query returns `None`, non is returned triggering the exception in BLL. Otherwise there is constructed the respectively Task and Result objects using their export functions from their respective internal services. Last a `MissionInDBWithRel` instance is create, a Mission with its relations, that is returned to BLL, where it is returned in a Responses.

Update

`update_done_by_id()` and `update_active_by_id()`
Updating the operational status of the a mission is by changing the boolean value of the attributes / entities `done` and `active`. Their logic is almost identical. Both function is called using a *PUT* function from CL with a `MissionUpdate` as request-body, containing the respective missionId and the new value. Then respectively `update_done_by_id()` and `update_active_by_id()` is called on a `RouteCRUD()` object moving the execution in DAL layer.
Each of the two DAL function with the same name, is making a query for the mission on the given id, with the appended function, here example with done: `.update(Mission.done:  up_mis.val)` updating the value directly in DB without loading it into memory. Both of the functions returns a string with the word "Success" in a Response instance (remember SQLAlchemy raises exceptions automatically in DAL). This is for moving the error-evaluation back to BLL. The BLL function does not return anything upon success.
`update_by_id()`
Uses a `PATCH` operation in CL, having a `MissionForUpdate` Pydantic class as request body. This class have all attributes optional, allowing only to parse the desired new values in the body. It is possible to update: name, swamId, done, active using this function. Be aware that is functions does have a performance penalty, which is therefore the two other functions above exist (they would be called more frequently). `update_by_id` is called on the `MissionService` object in CL. This BLL function is similar to the other onces, but does return `MissionInDB`. In `MissionCRUD.update_by_id()`, the mission is queried using the filter function. What is special in this query is the use of `.with_for_update()`, that sets a row level lock in DB, because we need to load the object into memory. This set a *FOR UPDATE* appended on the select statement, and forces all users trying to access the object to wait for the transaction to complete.
When the object is in memory, the native python function `setattr` is used to change any of the attributed in the input on the loaded object. After this, the object is committed to DB, opening the lock. `MissionInDB` is constructed with the new values and is returned to BLL function.

Delete

Calling the `DELETE` operation on the router-instance decoration. The router does not return anything, hence a `204` response code. A `response_class` (not model) is defined, with a FastAPI response, thats empty (see why: 4.10.1). The internal function takes a id, for the given tuple and a session (connection). A `MissionService()` instance is constructed calling the function `delete_by_id()` function in BLL. If the return value, from BLL, is not `None`,

the response handler would be called raising the exception contained in the Response (Remember exceptions have encoded their own status code etc.). Upon success nothing is returned.
In side the BLL layer function a `MissionCRUD()` instance is constructed and the function `delete_by_id()` is called, with the id as argument. If the return value is less than 1 and Exception is constructed and returned.

`update_by_id()`
The same Query construction is used on the Mission table. Called with a `filter` (Translated to `WHERE` in SQL). The `first()` function is appended. We are guaranteed uniqueness, so no need to keep iterating indexes. We are here actively relying on the ORM, utilizing the relationships, as it is more convenient code and do not course in more database calls. The Mission object is returned relationships (with the cascade settings: 4.8.4, defined in the Mission Mappable class, we can simply call the session objects `delete` function and parsing in the object. The mission and all orphans of the Task instance is marked for deletion during this process. When committed (or flushed), this takes persistent action in DB. This is an example where ORM really comes to its right with out much penalty. We are only loading a single mission into the applications memory. Note, that the tasks are not loaded in, because of the lazy-load in SQLAlchemy. 1 is returned to BLL, conforming to the concepts of other deletion function, but this is essentially superfluous.

4.10.3 Task Service

The internal Task service is responsible for the tasks, that dictates a route for each drone and is associated with a mission.

Mapping Class

Task mapping class can be found in the `models` directory (see appendix 9.3). The class uses `Geometry('POINTZ', dimension=3)` in the Column, to represent the location. The mission id is also a foreign key, which is to support the relations defined in Missions relations.

Pydantic Schemas

The schemas can be found in `schemas/task.py`. The `loc` (location) is represented by the earlier described `Point` (schemas/utils/geometries). All coordinates is validated already in CL layer, automatically handled by Pydantic. `dspec` (data specification) is used to tell the drone to which sensors to use. This is represented by a dictionary (exported in JSON) for flexibility and adding multiple instructions. A string is representing the attribute for now, because of issues with Pydantic JSON type (therefor `json.dumps` is used manually.)

Router - Control Layer

The router for task service can be found in `routers/router_tasks`. The construction of this router is the same as described earlier. All path operations construct `TaskService()` objects and call the respective functions, parsing the arguments. All functions, unless delete (that only returns the 204 status code upon success) is returning their response of type `Response` into `response_handler()`.
This router's *POST* decorations and path-operations is commented out, but kept for easy debugging. Tasks is inserted when the internal mission service is calling `TaskService()`, when creating a Mission.

Create

`TaskService` (BLL) contains the two functions `create_all()` and `create()`. `create_all()` is specifically used when a mission is created and many tasks is created at once. The functions is creating a `TaskCRUD()` instance and calling the function of same name, parsing a list of `TaskInDB` (Pydantic Schema)) objects. The function does not return anything.
The `create()` function is used when creating a single task. `TasksBase` schema is taken as an argument which parsed to the function of the same name, called on a `TaskCRUD` object. A Response object is returned containing `TaskOnCreate` after any BLL errors is checked.
The DAL function `create_all` is creating a list of Task's (mappable objects) using the static function `create_db_task()` (assuming that the PK is generated already). They are added to the Session using the `add_all()` function and the session is committed. The function does not return anything (Remember exceptions is automatically raised by Pydantic).

The DAL `create()` is creating a `Task` object from the `TaskBase` parsed. A `WKT` object is created from the Point attribute in TaskBase using `GeomConverter.geom_from_coordinates()` 4.10.5. Further `uuid` is generated. With these a Task object is created, add to the session and committed. A `TaskOnCreate` (schema) is constructed and returned. It only contains the Task's id (PK).

`create_db_task()`
Creating `Task` object where the `tId` is already generated. `GeomConverter.geom_from_coordinates()` 4.10.5 is used to compute the `wkt` object, before the Task is constructed and returned

Read

Tasks can be queried by mission id, task id and all of them with the respective functions `get_by_mId()`, `get_by_tId()` and `get_all`. The first two takes a id as argument and the last one takes `skip` and `limit`. All three of the functions is creating a `TasCRUD` instance and calling the respective DAL-functions of same names:
`get_by_mId()` is querying the Task table using the with the filtering on `missionId` should match the parsed id. `all()` is used to make SQLAlchemy return the result as a list. The list is iterated to create a list of `TaskInDB` schemas for export, done with calls to the static `create_task_export`. The list is returned. `get_by_tId()` is also querying the Task table using filtering, but on the primary key, which is also why the `first()` method is used as we are guaranteed only one occurrence. `create_task_export()` is again used to create an exportable schema object. `get_all()` is querying the table with the `offset()` to offset any given number of tuples. The `limit()` method limits the number. `all()` is also used here for list-return. Exactly like in get_by_mId the resulting list is made to a list of `TaskInDB` schemas, that is returned.

`create_task_export()`
Creates a `TaskInDB` Schema for to be returned in CL layer path operations. Persons with ORM- and SQLAlchemy knowledge would know it is possible to use a Python mapping with `**` operator to map between the mappable class and the Pydantic Schema. This can't be done that simple because of the spatial values (loc in Task) that needs explicit conversion to make sense in the applications.
The conversion itself is done with `GeomConverter.point_from_geom()` (4.10.5). Then the other values is simply parsed with the converted one construction the `TaskInDB` object that is returned.

Update

Logic to updates a Task is on purpose not implemented. This would alter the drones waypoints and routes there is selected from fixed positions in UI (In current version Towers).

Delete

The BLL function `delete_by_id` taking the Task id (PK) as argument is constructing a `TaskCRUD` object and calling the DAL function of the same name with the argument. The function does not return anything upon success (CL layer returns `204` on success). It the return value is less than 1, an exception is returned.
The DAL function is querying the Task table with the id using the filter method and execution the deletion in memory using the delete function. This returns the number of tuples deleted, which is returned to BLL. This is not synchronized (see explanation: 4.10.1). The changes is committed. The function do not return anything.

4.10.4 Result Service

The internal Response class is responsible for handling Results. As results is associated with the missions by a relationship the logic is separated and works independently without any calls between internal services.

Mapping Class

Results mapping class can be found in `models/results.py` (see it in appendix 9.3). The class does not introduce any new concepts. Refer to Missions or Tasks sections.

Pydantic Schemas

The schemas can be found in `schemas/results.py`. The schemas is `ResultBase`, `ResultInDB` and `ResultOnCreate`. The pydantic classes is reflecting a subset of the attributes with types from the mappable class. The only special attribute used is `Point` from `schemas/utils/geometries`.

Router - Control Layer

The router for results service can be found in `routers/router_result.py`. The router is almost identical with the Task Services'. Path operations creates instances of `ResultService()` and calling the respective functions in BLL. All services, unless the *DELETE* operation parsed the returned value from BLL into a `response_handler`.

Create

`TaskService` contains the function `create()` that constructs a `ResultCRUD` object and calling the DAL function `create()` with the `TaskBase` argument parsed from CL. The return values i checked not to be none before returned in a Response
The DAL function `create` is creating the `wkt` object from the Point in `TaskBase` and generating a uuid. With these a Result (mappable class instance) is created, added to the session and committed. A `ResultOnCreate` is constructed with the single attribute is has: the Results id. The object is returned to BLL.

Read

Exactly like Tasks, Results can be queried by mission id, result id and all of them with the respective functions, `get_by_mId()`, `get_by_tId()` and `get_all()`. Likewise, the first two takes a id as argument and the last one takes `skip` and `limit`. All three functions is creating `ResultCRUD` instances and calling the respective DAL-functions of the same names.
The DAL functions is identical to the onces described in task service (4.10.3). The only changes is the use of the function `create_result_export` to export, it is the corresponding result schemas used and that we are querying the Result table instead.
`create_result_export()`
Taking a Result from DB and construct a `ResultInDB` instance, replacing the `wkt` attribute with the converter one, using the `GeomConverter.point_from_geom` creating a Point.

Update

On purpose the it is not possible to change the Result's. This would nok make sense for this use case, and could potentially open a possibility for altering inspection results, exposing a security risk.

Delete

`TaskService` contains a `delete_by_id()`, making it possible to delete Result's by PK (id). The function does not return anything. A `RouteCRUD` object is created and the function of same name is called with the id as argument. The DAL function is querying the Result table with the id using the filter method and execution the deletion in memory using the delete function. This is not synchronized (see explanation: 4.10.1). The changes is committed. The function do not return anything.

4.10.5 Utils - Utilities

Utility module is extended in Missions Microservice.

geometryconvery.py

GeomConverter converts to and from `Well Known Binary` (WKB) type used in PostGIS (GeoAlchemy2) to represent geometry objects. The converter relies on Shapely [75]. Shapely is known to have some performance challenges in some cases, but not much well documented with valid sources. During the implementation it was tested moveing the conversion into the DB using `SqlAlchemy.func`'s in the query (Using ST_X, ST_Y, and ST_Z [17]), to get a WKB elements coordinates directly), to test the performance. The test-results can be seen in Appendix 9.4, with the conclusion that in our simple case, Shapely was by far superior.

`geom_from_coordinates()`
Converts from `Point` (Pydantic Classes) to a `WKT/WKB` object (can be interchangeably used in this implementation but be aware of limitations). The functions is constructing a Shapely Point. The objects `.wkt` attribute is returned.

`coordinates_from_geom()`
Concerts from WKB elements into coordinates (Positions class). Takes a `WKBElement` as arguments and uses Shapely's `to_shape` to construct a Point object. From here we can construct one of our own object validating Point objects. This is not strictly necessary, but makes it possible to have a standard type internally.

`point_from_geom()`
Construct a Point (Points class) from a wkt point. `to_shape` is used to construct an object, where the coordinates is called and used in the construction if the `Point` instance.

4.11 Drone / Logger Service

Drone/Logger services can be found in the directory 'logs' and consists of four internal services. The service integrates with the Mission and Estimator Services.

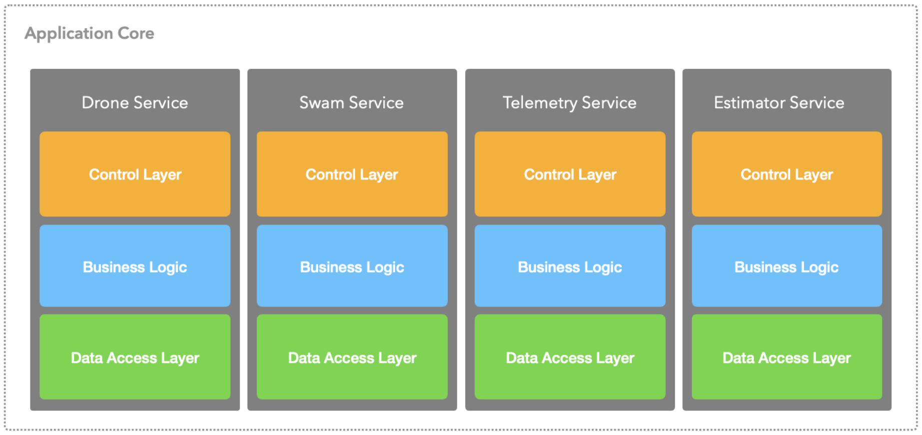


Figure 4.16: Drone/Logger Service - Internal Services

4.11.1 Telemetry Service

Telemetry services is handling logic around the Telemetries. Telemetry Services BLL and CL layer respectively implements `AppService` and `CRUD`. Therefore the DB (db) connection is available on the `self` object.

Mapping Class

The Telemetry Mapping class can be found in `models/telemetry.py` (see appendix 9.3). This class introduces the new type `ARRAY(FLoat)`. A Array is breaking with the idea of relational databases, but some systems offers the type these days, and PostgreSQL is one of them. It is use din the `ori` attribute that describes the orientation of the drone in that given time. It's very simple implemented, as the type can be parsed into the `Column` object and then everything works as expected.

Pydantic Schemas

The schemas reflects the Telemetry (mappable) class or a subset of it, fitted to the given use case. The special type `Orientation` is used to represent the drones orientation. This is just another pydantic class with three guarded values. The schemas can be found in `schemas/telemetry.py`

geometries.py Used the same custom internal geometry classes: 4.10.2.

Positioning.py Orientation found in `schemas/utils/positioning.py`. A class with three attributes: pitch, roll and yaw implementing the Pydantic BaseModel. Each of the attributes is guarded with Pydantic's `confloat`, validates that each values is within a valid range.

Router - Control Layer

The router can be found in routers/router_telemetry.py. The implementation is very similar to the other introduced routers, but we introduce a few twerks in this one. The `TelemetryService` contains a lot of logic, supporting the many endpoints from CL, that can share the same function signatures (declaration, arguments etc.). When the Router is calling *GET* or *DELETE* functionality in BLL, it parses a enum value determine on which parameter hence attribute should dictate which function to be called in DAL. Hence getting request have a decoration and path operation function uses a call to BLL (`TelemetryService`) function `get_by()` where the second argument is dictating which attribute to "get by".

```
@router.get("/telemetry/{mId}", response_model=List[TelemetryInDB])
    def get_telemetry_by_id(mId: UUID, db: Session = Depends(get_DB)):
        """
        Get Telemetry's by Mission ID
        """
        response = TelemetryService(db).get_by(mId, TelemetryService.toGet.mId)
        return response_handler(response)
```

Figure 4.17: Telemetry router endpoint calling BLL using enum value - CL

Same idea for the function `delete_by()`, but as with all delete-functions, no return of `response_handler()`. A Empty FastAPI Response (not our response instance) is returned with the status code instead. This is not trivial as this concept is weakly defined in FastAPI and a simple `204` return does actually return a non-empty response (Note Python's None is converted to "null", a non-empty response). This is a ongoing discussion [25].

Create

The Telemetry creation is orchestrated the function `create()` within BLL `TelemetryService` (constructed in CL path operation). The body of the request is by Pydantic parsed to a `TelemetryBase` instance and as a parameter to `create()`. It constructs a `TelemetryCRUD` object and call the DAL function of the same name, with the parameter of `TelemetryBase`.
The DAL function `create()` is converting the `Point` object in `TelemetryBase` to `WKT` object using the function `GeomConverter.geom_from_coordinates()`4.10.5. Together with the generated UUID (PK), the `wkt` string and the rest of `TelemetryBase` class attributes, a Telemetry object is constructed. The object is added to the session and committed to the database.
Lastly a `TelemetryOnCreate` is created (only contains the PK) and returned to BLL function.

Read

Telemetries can be queried using mission id and drone id. As explained in the Router section, they call the same function, `get_by`, from CL Layer.
`get_by()` a simple `if/else` structure comparing the enum-values. (Enums are defined as inner classes to the service class). The body of the `if` or `else` statement is constructing a `TelemetryCRUD` object and calling the respective DAL function. Either `get_by_mId()` or `get_path_by_dID()`. A custom `Get()` exception if the DAL functions returns `None` instance. Upon success the return value is returned in a `Response`.

`get_by_mId()`
Using the Query api, the Telemetry table is queried using `filter()` to filter by the mission id and the `all()` function is again used to get a list as query-result. Without this, a tuple containing the result is returned. The use

of `all()` returns a penalty on performance to some degree, but comes with the easy handlings of empty returns and edge cases.
The result is iterated and manually mapped from Mappable classes to schemas using `create_telemetry_export`, constructing a list of `TelemetryInDB`, that can be returned in BLL to CL and serialized without further, as Pydantic handles it.

`get_path_by_dID()`
The `pos` attribute on the Telemetry table is queried using a filter to match tuples where the drone id match. The result is a list, which is parsed to `create_points_export` creating a list of schemas from the mappable objects. The list is returned to BLL.

`create_telemetry_export()`
This static function is a part of `TelemetryCRUD`. It constructs and returns a `TelemetryInDB` parsing in the attributes from the mappable class, with the `wkt/wkb` object converted into a Point using `GeomConverter.point_from_geom` for the `pos` attribute. Further for the `ori` attribute, a `Orientation` instance is created.

`create_points_export()`
This static function is a part of `TelemetryCRUD`. It constructs and returns a `TelemetryInDBPath`. A schema only containing a list of Point instances. The parsed list is iterated creating Points with `GeomConverter.point_from_geom` appending each of them to a list that is returned.

To be able to estimate the Estimator Service would need the last 2-3 positions of a drone. Therefore the the CL layer contains a dedicated endpoint for this, calling the BLL function `get_last_n_positions_by_dId`. This function is not a part of the structure above, as it contains one more parameter: the number of positions requested. The function uses the well known structure of these functions and call the DAL function `get_last_n_positions`

`get_last_n_positions()`
This function queries the attributes: `pos`, `time_created`, `mId` and `dId` on the Telemetry table. The query is filtered by drone id. The `sqlalchemy.desc` is used to order the result in descending order (SQLAlchemy's compiler produces a `ORDER BY SQL construct` [77]). `limit()` is used to limit the number of results and `all()` makes the result a list. `TelemetryCRUD.create_points_time_mid_export` is used to construct a list of `TelemetryInDBPointTimeMId`

`create_points_time_mid_export()`
The list produced by SQLAlchemy from the query result is iterated, constructing `TelemetryInDBPointTimeMId` objects that is appended to a list. We can't use the Python mapping operator `**` as the positions needs to be converted from `wkb` to a Point. This is done with `GeomConverter.point_from_geom`

Update

Telemetries can't be updated, it does not make sense in this use case.

Delete

A Telemetry can be deleted on its PK (`tId`), mission id and drone id. The three dedicated path operations in CL layer calls the BLL function `delete_by` (See CL section above). A `if/else` structure compares the enum-values. The respective DAL function is called on a `TelemetryCRUD` object constructed. Nothing is returned if the number of mathcing tuples is greater than 1 else an expception is returned.

`delete_by_mId()`, `delete_by_dId()` and `delete_by_id()`
The three DAL functions is almost identical, with the only difference being the `filter` methods parameter. They all makes a query on the Telemetry table, filter by their respective id. Using the `delete(synchronize_session = False)` (see details:4.10.1) method, the objects are deleted. The changes are committed to reflect the changes in DB. They return the number og tuples matching.

4.11.2 Drone Service

Drone Service can be found in `service/drone_service.py`. The service is responsible for the representing the drones in across the Missions- and Drone/Logger microservices Be aware that the foreign key (FK) constrains, about existing drones is guarded (by SQLAlchemy, because of the defined relations) withing the Drone/Logger services. But currently, the UI and VRP Service does not rely on service to get the available drones. This means that this services contains some work-arounds in the creation. When all services rely on the internal drone service for drones, then only valid drones can be picked in the UI and hence be in missions attributes (This bridging is not a part of the thesis).

Mapping Class

Drone mapping class can be found in the `models` directory (see appendix 9.3). The class only consists of the primary key, `dId`, that is of string type. In the future, when the VRP service is properly integrated, this can be changed to UUID. For know the string provides a requested flexibility. Further it defines a one-to-many relation to estimations and a two bi-directional estimations, to swams and telemetries. They are defined for supporting use of relations, if wanted by future implementor.

Pydantic Schemas

The schemas can be found in `schemas/drone.py`. The schemas defined is `DroneBase`, `DroneInDB` and `DroneOnCreate` which is a alias for `DroneBase`, to keep the naming conventions. `DroneBase` consits of the dId attribute and `DroneInDB` extends it, by adding the relations.

Router - Control Layer

The Router in the drone Service, does not introduce any new concepts. The only difference is the objects created in path operations is `DroneService` instances, which is implementing the `AppService` class to obtain the Session object.

Create

`DroneService` contains a single `create()` function. This is called from CL with a user-chosen string value as dId for now. The function construct a `DroneCRUD` instance and calls the function og the same name in DAL. The return value is checked to not be a `None` value. This makes a robust seperation between the two layers, and a change that introduces a None value would course a `Create()` exception to be thrown. DAL `create()` function is constructing a Drone object, adding it to the session and committing it to the db. A `DroneOnCreate` object is created and returned to BLL.

Read

a Drone can be queried by id with the function `get_by_id()` that return the a drone with the Swams it is participating in. The function call the DAL function of same name, and returns it in a Response object.
`get_all()` have the parameters `skip` and `limit`. It calls the DAL function with the same name on a `DroneCRUD` object, parsing both parameters. The return value here is allowed to be a empty list, this is specifically handled logically (In Python `None == []`).Drones is returned with their respective swams
`get_all_ids` have the parameters `skip` and `limit` and calls the DAL function of the same name. The response is a list of only drone ids or `[]` as just described this is allowed.
`get_by_id()`
The function is querying the Drone table, with the `.joinedload(Drone.swams)` and `.joinedload(Drone.estimates)` (described in Mission Service), joining the Swam and Estimate table. The defined relations between Drone Table and the Estimator and Swam table, means no keys need to be specified. Further the `filter` method is used to find matched on the `dId` and lastly `.all()`. The query-result is parsed as argument to `create_drone_export` which result (a list of `DroneInDB`) is returned to BLL.

`get_all()`
The function is querying the Drone table, with the `.joinedload(Drone.swams)` and `.joinedload(Drone.estimates)`, with a `.skip()` and `.limit()` function before the `.all()`. This produces a list of Drone objects.

The list is iterated with each object parsed to the `create_drone_export`-function and appended to a list, creating a list of `DroneInDB` objects that is returned.

`get_all_ids()`
The function makes a query on the `dId` attribute on the Drone table. Further `.skip()`- and `.limit()` functions before the `.all()`. The query-result can be returned without further.

`create_drone_export()`
Taking a Mappable Drone object and returns a `DroneInDB` schema. First the drones swams is iterated to create a list of `SwamOnCreate`. This is done with the static function `SwamCRUD.create_swam_export()`. The same is done with estimations.
The `DroneInDB` is constructed with the two lists, the drone id. The object is returned.

Update

As the drones do not consists of more than the id, a update does not make much sense, as a new one can simply be created. If we allowed the changes of PK, we can allow the users to introduce anomalies and clashing keys.

Delete

The BLL function `delete_by_id` calls the DAL function of the same name. If the returned value from DAL is higher than 0, nothing is returned. Otherwise an exception is returned. The DAL function is querying the Drone table with the `.filter()` function to obtain the match. The object is deleted using the `.delete(synchronize_session=False)` and comitted to DB.

4.11.3 Swam Service

The internal Swam Service is responsible for the Swams. They keep track of a tribe of drones associated to a mission.

Mapping Class

The mapping class can be found in `models` directory (see appendix 9.3). As described in the Database Design section, this class have a auto-incrementing id. Essentially a PK is not needed for this relation, but SQLAlchemy forces this. Otherwise a swam consists of a swam id and a relation to Drones table.

Pydantic Schemas

the schemas can be found in `schemas/swam.py`. The schemas defined are `SwamBase`, `SwamInDB` and `SwamOnCreate`. `SwamBase` contains the two attributes swam id and a list of drones id's. `SwamInDB` is extending the first one with the primary key. `SwamOnCreate` have a single attribute. The swam id.

Router - Control Layer

The router in Swam Service does not introduce any new concepts. Instances constructed in path operations is of type `SwamService`, which is implementing the `AppService` class to obtain access to the Session object.

Create

`SwamService` contains a single `create` function, that is called from the CL layer. The function takes a `SwamBase` as argument (from request body). The functions construct a `SwamCRUD` instance and call the DAL function of same name.
The DAL function is creating a `Swam` object by iterating the drone ids in the `SwamBase`. The objects are appended to a list. The list is added to the Session using `add_all()` and committed afterwards using `commit()`. A `SwamOnCreate` object with the swam id is created for return.

Read

Swams can be queried by mission id and drone id. The two functions BLL `get_by_dId` and `get_by_sId` is called on `SwamService` instances in CL. Each of them call their respective functions of same name in DAL. The response value is checked to be not to be None (empty list allowed), before they are returned in `Response` object.

`get_by_dId()`
Make a query on the Swam table, using the `filter()`- function finding swams containing the drone id. A `.all()` function. The list of swams is iterated creating a new list of `DroneOnCreate` objects using `create_swam_export`.
`get_by_sId()`
Getting a swam specific swam, meaning quering all the tuples containing the specific swam id. Therefore the Swam table is queried with the filter for matching sId's. The query is returned as a list because of the appended `.all()` method.
Using `DroneCRUD`'s `.create_droneBase_export` function, the results is converted to a list of `DoneBase` objects, which is returned.
`create_swam_export()`
Constructing and returning a `SwamOnCreate` on create object from the mappable swam object.

Update

Swams are created from the VRP response, hence the selected drones. If the swams would be updated there would be no guarantee, the tasks for the designated drone, would not be done - further the VRP close to optimal solution would maybe not be optimal anymore.

Delete

`delete_by_id()` creates a `SwamCRUD` object which the DAL function of same name is called with the given id as parameter. Nothing is returned from BLL upon success otherwise an exeption within a Response object is returned. CLL returns a `204` status code from CL upon success. The Swams table is queried for matching tuples to the id. They are deleted deleted using the same strategy as in other internal services. The changes are committed to the DB for persistency.

4.11.4 Estimator Service

The internal estimate service is responsible for orchestrating a estimate positions for a specific drone. The service get a request for a estimate on a specific drone (on id). The services is then querying for the last three positions using the telemetry service and sending a request with those the Estimator Microservice. The response for the service is saved in database for later analysis before returned to the UI.

Mapping Class

The Estimate mapping class can be found in `models/estimate.py` (see appendix: 9.3). The implementation does not introduce any new concept, not already covered.

Pydantic Schemas

The schemas for Estimate can be found in `schemas/estimate.py`. The schemas is `EstimateBase`, `EstimateInDB`, `EstimateOnCreate` and `EstimationPoint`. Like the other schemas they represent the attributes (or a subset of) the mappable class.

Router - Control Layer

the router for estimator service can be found in `routers/router_estimate.py`. The *GET* operation is the creating operation for BLL and CRUD, ad its orders the creation on demand.

Read (Create)

On the `EstimatorService` instance in CL the `estimate_manager` is called, with the given drone id as argument

`estimate_manager()`
Manages the information flow. Collects the last positions from the database and requests the estimator service.
The function `get_positions()` returns a tuple the a list of `TelemetryInDBPointTimeMId` and the number of points.
If the number of points returned are 0, then a `Get()`-exception with the relevant information returned.
When there is exactly 1 position, there is still not enough positions make any estimations. The only known location in a EstimateBase Wrap (for contentious) is returned. This also means that it possible to get a response already after first reported Telemetry.
If there is 2 or more telemetries, hence positions, a proper estimate is possible. A request-body returned from the function `construct_request_body`, taking the drone positions as argument. The body us used in the call to the Estimator Service, orchestrated by `call_estimator`, taking the Estimator URL (obtained from the config using `get_config_estimator()`) together with the request-body. If the response from estimator service is none, or the call not succeeds, a custom `Request` exception is returned, with the relevant information.
The returned value is serialized by the Estimator Service using a `EstimationPoint` object schema, hence we can simply use python mapping `**` operator to deserialize the response into an `EstimationPoint` object again.
A `EstimateBase` object is constructed using the new estimating point and dId and mId from one of the `TelemetryInDBPointTimeMId` objects. The object is given as argument to a `create()` DAL function called on a constructed `EstimateCRUD()` object.

`get_positions()`
Call the `TelemetryCRUD.get_last_n_positions` function with the dId and the integer value 3 as argument. We want up to 3 `TelemetryInDBPointTimeMId` schemas back. The response is checked for a `None` value returned. If so a `None` is returned coursing `estimate_manager` to return a `Get` exception (that will be raised by response_handler).

`construct_request_body()`
Taking a list of `TelemetryInDBPointTimeMId` and creates and returns a list of serialized objects `TelemetryPointAndTime` using the FastAPI's `jsonable_encoder` (Essentially the its produces a `json.dumps()` result of the class. All Pydantic classes can be made to a dictionary using `.dict()`. This result is then serialized).

`call_estimator()`
Executes the actual call to Estimator Service. The functions is called with the URL and the request body (organized in `estimate_manager()`). The call is performed in a safe block. This meaning a `try`, `except`, `else` structure in Python. Using `requests.post`, with the url and body, parsed directly from parameters, a request is send to Estimator Service. If the exception is triggered the function simply returns `None`, coursing the `estimate_manager()` to return a exception type `Request`. In the implementation i can be seen that the logic to extract the exact exception from `exceptions.RequestException` is present, but out commented. We don't want to expose any details on how our distributed system communicated through endpoints, and hence this is only kept for debugging purposes. In general parsing every AppException within a Response to `response_handler` would course the immediate raising of the exception.
On the other hand, if the request is successful, the contents is decoded to 'utf-8' and converted into a Python dictionary using `json.loads` before returned.

`create()` **(DAL)**
Taking a `EstimateBase` object and returns a `EstimateOnCreate`. A UUID is generated and the Point in `EstimateBase` is converted using `GeomConverter.geom_from_coordinates()`. Together with the rest of the attributes in `EstimateBase` a instance of the mappable Estimate is constructed. The object is added to the session and committed for persistency.
Last a `EstimateOnCreate` object is constructed and returned with the generated id as the only attribute.

Update

Updates is intentionally not a possibility. A estimate is generated from the telemetries reported by the drones, as they can't be updated/changed neither can a estimate. It does not make sense, and the possibility, could foster a future operational security risk.

Delete

Estimations can be deleted on their primary key (`eId`). Because of the relationship between mappable class' Drone and Estimate it is possible to get all a drones estimates (be aware that doing so would load them into memory because of ORM, this is not the most efficient strategy). The lifetime of the relations is not decided yet, hence we focus on a simple implementation to show the concept, that supports a easy extension to support the chosen strategy in the future.
The BLL layer `delete_by_id()` is called from CL path operation on a `EstimatorService` instance. It creates a instance of `EstimateCRUD` and call the DAL function of same name. The function makes a query on the Estimate table using the `filter()` function finding the tuple matching with the PK. This is deleted using the `synchronize_session=False` deletion strategy. BLL layer returns an exception if a value above one is not returned from DAL.

4.11.5 Utils - Utilities

geometryConverter.py

The Drone/Logger Services uses the `GeomConverter()` from `utils/geometryConverter.py`. This is the same module as used in the Mission Service. See description: 4.10.5.

4.12 Estimator Service

The Estimator Microservice is responsible for compute the drones estimated position, from its previous reported positions. The service is integration with the Drone/Logger Service, which request the service. This Microservice does only have one internal service, the estimator service. The service extends the base directory structure with a `parers` directory. This service do not rely on a database, hence do not implement any database abstraction. The internal service does not implement the `AppService` nor `CRUD` classes.

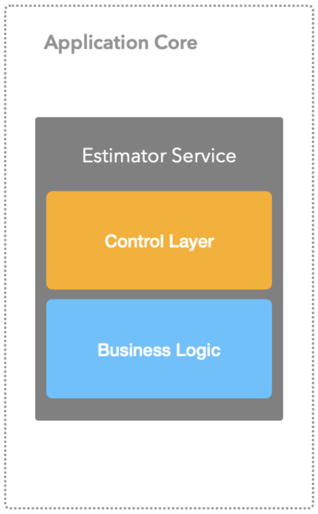


Figure 4.18: Estimator Service - Internal Service

Pydantic Schemas

Estimator the same `Position` and `Point` schemas a used in Drone/Logger- and Mission Service. They can be found in `schemas/geometries.py`. See description: 4.10.5. Further, a new schema is defined, `EstimationPoint`, which only have a `Point` as attribute. This schema is used to serialize and return the result. It can be found in `schemas/schema.py` together with the already introduced (in Drone/Logger Service) `TelemetryPointAndTime`. The latter one is used to map the request into Pydantic class.

Parser

Found in `parsers/parser.py` module is the function `to_vector_list()`. The functions converts the list of `TelemetryPointAndTime` from the request-body to a list containing objects of `VectorPosition`. This is simply done by iteration over the list of positions, constructing a vector (list) of the coordinates within the Point object. (Using the function `get_Cords()` found in module). Together with the Time (`time_created`) attribute, the vector is used to construct `VectorPosition` object.

Router - Control Layer

Fond in `routers/router.py`
The service only exposes a single endpoint defined CL. There is not introduces new implementation concepts.

4.12.1 Estimator Service

Can be found in `services/estimator_service.py`.
The orchestrating function `position_manager` is called on a constructed `EstimatorService()` object in CL. The body of the request is parsed as argument to the function from path operation.

`position_manager()`
The manager function is responsible for coordinating the estimation internally. It it called with the positions parsed as argument. Using the `to_vector_list`-function, each of the positions is converted to instances of `VectorPosition`.
The service support the differentiating between a drone in acceleration or constant speed. Acceleration needs 3 points in space whereas constant speed evaluation need 2.Therefore, the number of positions given as argument is evaluated `if/else` structure. If the number of positions is less than or equal to 1, a `AppException.Estimate` exception is returned. Be aware that this structure is made to encapsulate the service better from the other onces in the distributed system. In `LiMiC1.0+`, the in-between service communication is validated and nothing is assumed about types and structure. This prevents future changes in one service to course errors inside another (a validation error would be thrown by Pydantic when validated in path operation function). The current implementation of Drone/Logger Service does not even call the Estimator Service if there is not enough Telemetry tuples available in DB.

2 positions: With 2 positions available, a constant speed estimation can be calculated. The function `constant_speed()` is called with the positions as argument.

3 positions: With 3 positions available, we can evaluate if the drone is accelerating. The function `accelerating_speed()` is called with the positions as argument.

Both functions return a `EstimationPoint`, that is returned in a `Response`.

`constant_speed()`
The two points is sorted by the `time_created` attribute of the objects, with the last point reported first in the list. `sort_positions()` is responsible for this. The points are parsed `EstimatorService.constant_speed_evaluation` in specific order (This could be done within the called function instead. But for consensus between 2 and 3 points computation we do it here. Done within the called functions, would mean it will be done 2 times in the accelerating case). The function returns the estimated positions, which is parsed to the static function `EstimatorService.export_estimation_point`, to construct a `EstimationPoint` that is returned.

`accelerating_speed()`
The three points is sorted using the same strategy as just described.
The three points is extracted on their positions and is in specific order parsed to the `isAccelerating()` function with a threshold value from the `core/config.py` file (called with `get_acc_threshold()`). The threshold is a percentage value, that the accelerating speed need to surpass. This is implemented to take care of the natural impacts when flying (wind gusts etc.), that could introduce "fake acceleration" that the drone would prevent by slowing down immediately, and hence not accelerating. `isAccelerating()` returns a boolean value, determine if acceleration estimation should be used. If so, the `EstimatorService.accelerating_speed_evaluation` function is called with the

positions in a specific order. The response from this is parsed to `EstimatorService.export_estimation_point`, constructing the serializable `EstimationPoint` that is returned.
If there is no acceleration determined, `constant_speed()` is called instead.

`EstimatorService.constant_speed_evaluation()`
The function computes the time from the current point to the future point. This is is the time now (`datetime.now()`) subtracted by the time from the current point. The difference in seconds with between the two points is then parsed to the `estimate_const_vel()` function in the estimator module (`utils/estimator.py`) with the two points themselves. The return value is returned.

`EstimatorService.isAccelerating()`
Determines from 3 points in space if the drone is accelerating. Takes tree `VectorPosition()`'s and a threshold and return a boolean value. The accelerating must be higher than a certain threshold in order for the function to return `True`.
To test if there is a acceleration, we get the speed vector between the first and second point and the second and third. The norm of the vectors is computed using `np.linalg.norm()`, to get a scalar-value. Then the percentage formula for calculating the change from one value to another: $(new-old)/old \cdot 100$ is used, calculating the difference in percent. The number is checked to be greater than the threshold (with absolute value, as negative acceleration is possible). Depending on that, `True` or `False` is returned.

`estimate_accelerating_vel()`
Found in `utils/estimator.py`. Acceleration is defined by $\frac{\Delta v}{\Delta t}$: Difference in velocity over change in time.
The functions is parsed 3 points and the future time. Two speed vectors is computed from respectively the first and second position and the second and third. Δv-vector is calculated by subtracting the two vectors. This is actually the accelerating vector, as the speed is: speed - last speed. This vector is multiplied with the future time and added to the last known position. This is the returned estimate.

`EstimatorService.export_estimation_point()`
The three coordinates is extracted from the list and used in the construction and return of `EstimationPoint`, that contains a `Point` object.

4.12.2 Utils - Utilities

estimator.py

The estimator module can be found in `utils/estimator.py`
The estimation uses simple positions represented by vectors in three dimensions. Using vectors makes the implementation more simple and we do not need to use any trigonometry. Calculation a approximate position on Earth, requires taking into account the height from the surface of Earth. As the Earth is spherical, we need to take the bearing into account, to calculate the height. But before we get to involved with this theory, because the earth is actually slightly ellipsoidal, the error-rate for a spherical model is actually 0.3% [4]. This is actually quite a lot for the use of drones. Hence we keep the this simple, and rely on the drones new reported values to "nullify" the potential errors

`estimate_const_vel()`
The function uses what is called "The Speed Vector" in the implementation. Think of it as a direction vector. This vector is computed by `get_speed_vector()`. The returned value is multiplied by the parsed future time and added to the current position's coordinates.

$$EstimateVector = currentVector + (futureTime \cdot speedVector)$$

Figure 4.19: Estimate Vector - Constant Velocity

`get_speed_vector()`
Essentially the speed vector is a direction vector divided by the time difference, getting value of change per second (times unit is second).

$$V_{V1\_V2} = V2 - V1$$

Figure 4.20: Direction Vector Calculation

$$V_{Speed} = \frac{V_{V1\_V2}}{V_{Time\_Difference}}$$

Figure 4.21: Speed Vector Calculation

In the implementation, the two coordinate-lists (that is parsed as arguments) is used in the creation of two np-arrays. This allows us, to simple subtract the last from the current. This gives us a direction-vector pointing in the flying direction, with values in in size depended on their creation-time. For "normalize" the values against a single second, the vector is divided by the time difference. This is done by constructing a third np array, with the time difference value on each index (vector is always of size 3). It is very likely, that we would need to divide with 0, as some indexes potentially contains 0 in the vectors. As this is mathematically illegal operation, a normal division would crash the code. The context manager `np.errstate` [42] used to allow and ignore the illegal operation. Within the context manager, `true_divide` is sued to divide the two vectors. Line 3 handles any `inf` results and set them to 0, likewise the `nan_to_num()` [43] to replace any `NaN` with 0.

```
1           with np.errstate(divide='ignore', invalid='ignore'):
2               speed_vec = np.true_divide(v1_v2, time_dif_vector)
3               speed_vec[speed_vec == np.inf] = 0
4               speed_vec = np.nan_to_num(speed_vec)
```

Listing 1: Context Manager - Division by 0 return 0

After this calcualting the vector can be returned.

vectorposition.py

Contains the class `VectorPosition`, that offers functionality used in computing the estimation. The class's attributes is a Time (`time_created`) value and the coordinate, a list: `[x,y,z]`. The class implements the two rich comparison methods [53]: `__eq__` and `__lt__`. They are used to define what equals or less than means. In this implementation the logic is around the time-created attribute. This allows us to use the `sorted()` function in the `EstimatorService()`.

Test Cases

Figure 4.22 show a use of a constant speed evaluation. The full examples, including with constant acceleration, with requests, and another example with constant acceleration can bee seen in appendix 9.6

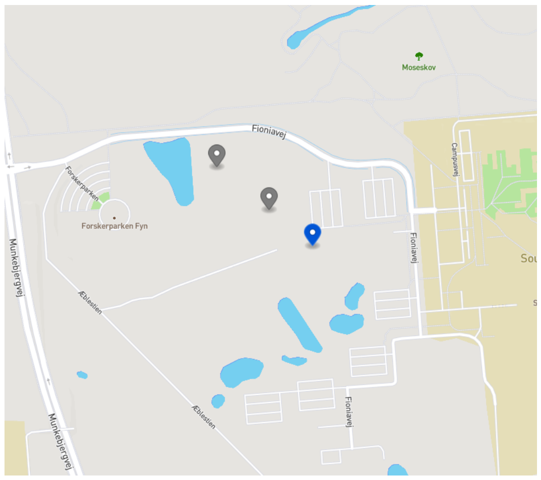


Figure 4.22: [Constant Speed] Estimation Point (blue) - Internal Services

Chapter 5

Conterization

The LiMiC1.0 paper states the use of microservices in a Kubernetes environment using containerization powered by docker.

5.1 Docker

Docker Containers is declared using a `Dockerfile`, that becomes a container during runtime on the Docker Engine. The big benefit is that each application can run in an isolated environment that run uniformly independently of deployment environment, staging, production etc. Why are we not just using traditional virtual machines(VM)? Containers and VM's have the same resource isolation and allocation benefits, they do not work in the same way. Containers virtualize the operating system where the VMs have their own operating system and virtualize the hardware using a hypervisor. This means that the containers is quite more lightweight (as the do not all have a whole OS kernel each), efficient and portable. Containers are an abstraction in the application layer. Code base and dependencies required is packaged together for each application. Because of the OS kernel abstraction multiple containers can share it, running an isolated process user space. The Docker Engine can run almost everywhere today, Linux, windows server, serverless etc.s [10]

Dependency Management

Dependency management is controlled using `requirement.txt` files. This is mainly because they are very easy to use and change during development. For production the use of `Pipfile`'s should be considered, where every installed package is specifying a hash. This means malicious infection in packages can without any check be installed into the dependencies. The hash of the installed package used in Pipfiles `Pipfile.lock` secures that we also we know that the version number that we specified always corresponds to the same package in the lock file.
The existing LiMiC1.0 using one global Pipfile, this is most likely not a desire when the these services is introduced, as they have a lot more dependencies. This state leave easy reorganizing for the DevOps team.

Docker Images in LiMiC1.0+

Each of the three applications have their own `Dockerfile`. All three of them are very close to be identical. De they be seen in the root-directory of each microservice
All `Dockerfile`'s is initializing the new build stage using `python:3.9-slim-buster` as Base Image, this is set using the FROM instruction. The EXPOSE instruction is used in the Estimator Docker, that instructs the container to listen on this port at runtime. This is set using Compose for the other two services. Next the two environment-variables `PYTHONDONTWRITEBYTECODE` and `PYTHONUNBUFFERED` is set to the value 1. The first one prevent python from generating `.pyc` files in the container and the latter turns of the Python buffering, which makes easier container logging (these a primarily set for conforming to same behavior as the existing Docker images).
All Docker images have a `WORKDIR` defined. If it is not set by the implementer the Docker Engine creates one. The instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instruction. [6].
Next the Python dependencies are installed. This is done by installing pip and upgrading pip to latests versions. Next, COPY is used to copy the `requirements.txt`-file into the working directory. From here it is installed using pip. We do this copy seperate from the copying of the code, because docker caches in layers that, can be reused. This seperation defines a layer for the requirements alone (that dosn't change often.) [68]

Next, the code is copied into the working directory. This is places intentionally near the bottom of the file for optimized build times. This is containing the the code, which is changing most frequently, and all steps after this is invalided in the Docker Cache. This is the end of the file, for the containers going into a multi-container setup. For stand-alone containers, a `CMD` instruction is defined. Simply with the same parameters normally used in terminal.

Building and running Estimator Service

- build: `docker build -t estimator`. `-t` is a tag flag, that makes is possible to refer this image.
- run: `docker run -p 8080:8080 estimator`
- run for test locally: `docker run --network=logs_default -p 8080:8080 estimator` (using test docker-network)

5.2 Docker Compose

When our applications is dependent on a database, we define them using Docker Compose, which is a tool to define multi-container applications. With one `YAML` file, the whole application stack can be defined. This file can with a single command spin up all services and tear them down [7].
A docker-compose file consist of a *Services* and a *Volumes* section.

***Services*:** Services sections is defining the individual services. In our cases this app-service and a db-service.
app

`build .` Setting the build context. This is a path to a directory containing a Dockerfile

`command` Command to start the container (Therefore we don't add it to the dockerfile)

`volumes` All the projects code is mounted at the relative path ., as the respective working directories (*/missions* and */logs*) (even if they already exist). This mount, makes sure that the changes to the code during local development is mounted inside the container on startup. They is being permanent into the container when rebuilding.

`ports` Mapping ports in HOST:CONTAINER format.

`depend_on` Defines dependency between services. Hence the service db would start before app.

db

`image` Base Image. `postgis/postgis:13-3.1-alpine` is used for PostgresSQL with PostGIS extension

`user` The database user-name.

`volumes` To persist the data beyond the life of the container we configure the values. We are binding *postgres_ data* to the directory */var/lib/postgresql/data/* in the container. This here done with *short syntax* and a named volume [8].

`ports` Mapping ports in HOST:CONTAINER format. The port is mapped to a different port in the Missions Compose files, to prevent the ports clashing when spinning up locally (could also have been opposite).

volumes

`postgres_data` Named volumes must be listed under top-level volumes key.

Create PostgresSQL user and Enabling PostGIS
Using `docker exec psql` would open a PostgreSQL terminal where a user can be set up. In the current implementation is is just a standard user, as seen in the `core/config.py` file. It is possible to configure this within the docker file [9]. As PostGIS a an optional extension it must be activated din each database. The: `CREATE EXTENSION postgis;` command is enough for the use-cases in this implementation [47].

Debugging local Docker-Network
This is not relevant for deploying of the services, but is was necessary to set up af docker-network, to enable communication between the containers. Multi-containers set up their own network upon start-up and single-container files join the docker default network.
The concept to instruct one of the multi-container applications (Missions is responsible) to create a user-defined network that Drone/Logger and Estimator Services join upon start.
The additions to the docker files (docker-compose.yml) is pointed out in the file by the comment *# DEBUGGING NETWORK*, so they are easy to remove for Kubernetes deployment.

missions/docker-compose.yml
Inside the top-level 'services' key, `networks` instructs the services app and db to join the networks, here only one: LiMiC. The top-level networks is defining the network called LiMiC, which runs using the bridge driver.

logs/docker-compose.yml
Inside the top-level 'services', `networks` instructs the services app and db to join the networks, here only one: missions_LiMiC (LiMiC from missions get pre-pended the container name). The top-level networks key is defining the network, with the under-key, external set to true, telling Docker not to create a network but join it instead.

Building and running Mission- and Drone/Logger Service

- build and run: `docker compose up --build`.
- run: `docker compose up`

Se the appendix,9.1, for the terminal log results upon start up. Here it can be seen that the database starts before the application. Further it can be seen that the Uvicorn webserver starts right before the application.

5.3 PostgreSQL in Kubernetes

LiMiC runs in a `Kubernetes` environment and hence we need to decide wether to tun the databases managed in this environment as well, or to introduce more operational involvement with handling the database outside the environment. In general the data layer hasen't got much traction with conterizations. This is not surprising comming from our analysis, that traditional DB is hard to scale horizontally and distributede systems is very much into that [2].
Having a database running in Kubernetes is closer to a managed solution (as the rest of LiMiC) and we can rely on kubernttes autoomatiion to kepe the database running. But, as the database application containers (the so called pods) are transient/mortal, meaning a higher change of restarts or failovers than in a traditionally hosted or fully managed database. Further there are database specific tasks, as backups, scaling etc. that are different due to the abstrations introduced by containerization. It would be easier to run a database on Kubernetes if concepts like sharding, failober elections and replications was build into the systems. Furhter databases that is working with transitens and changin layers in a application is more applicant fit for Kubernetes. Last, application with a Asynchronous mode of replication opens for data loss, because of transaction might be comitted to a primary database but not the secondary onces.
Using Kubernetes StatefulSet, data can be stored on presistents volumes, decoupling database applications from the persistent storage. This means the data is persistent through a pod-mortality. Further when a pod is recreated it is given the same name, securing a conistent endpoint-connection within the system.
When running databases as `PostgreSQL` that does not fit the model for a kubernetes friendly database, it should be considereed to use Kubernetes Operators (or similar). Operators wrap databases in additional features, that will help spin them up and perform maintenance as backups and replications. `Crunchy Data` or `Patroni`(Zalando) is such a operators for PostgreSQL. These operators use Kubernetes custom resources and conotrollers to expose apllication-specific opereations thorugh the Kubernetes API.
As it is not trivial and with an exact solution, adding an extra complexi to run RDBMS in Kubernetes, it is totally posible [2].

Chapter 6

Testing

The tests is done with the suggeted FastAPI setup. This relies on the Starlette (which FastAPI is built on top of) open `TestClient` [62] and the framework `pytest` [23]. Each of the three services main directory contains a directory called `tests`. The file `test_main.py` in each of them are initializing the test setup. other files is simply a seperation in separate classes (this is not a requirement, in pytest everyting with `test_` prepended are executed).

6.1 test_main.py

This files setups the dependencies and constructs and initializes the needed objects. They can be found in the `test` folder in root of every microservice.

6.1.1 Test without database

This setup is used in the Estimator Microservice.
The `app` variable, containing the application, from the `main.py` file is imported. the variable is used as constructing argument to the `TestClient()` object. Constructing the client, another arguement: `raise_server_exceptions=False` is used. This allows internal 500 errors to propagate to 500 errors (As if called from outside client). The `TestClient()` object can be used in the same way as `request` can be send to the service. Se example in appendix 9.5

6.1.2 Test with database

This setup is used in the Mission - and Drone/Logger Microservices.
A database url needs to be defined. In staging it can be the same, as the one defined in config, but at some point a dedicated database for test is preferable.
Then we are going to create a database engine using `create_engine` and create a Session object using `sessionmaker`. We could make a function in `database.py` that can be called to get these. For encapsulation and readability is it chosen to keep the test as a "add-on" and not directly integrated in the application. Further `override_get_db` is defined. A function that like the one in `db/sessionconnect.py`, that returns a session from the `sessionmaker` on call. Because this dependency is already set during the construction of the production/staging application (when importing app), we overwrite the `get_DB` (from `sessionconnect.py`), using the `dependency_overrides` function on the `app` object.
Lastly, the `app` variable, is used to to construct the `TestClient()`. Just like without the DB, the second argument is `raise_server_exceptions=False`.

6.2 Types of Tests

The overall testing strategy is focussing on the endpoints of the services. Some of the return values is `asserted` and used as values later in the running test (PK's mainly). As you probably should separate testing and developing, can look at you outgoing Pydantic, Response models test. Pydantic would make a error return, if the returned value (serialized) do not meet the validating requirements when constructed. The Pydantic models in the services is in general very strict.

The tests written is End-to-end and integrations-tests. As microservices includes more moving parts than a monolith application. End-to-end tests is a good tool to add coverage of the gap between the services. Further, end-to-end tests allows the architecture to evolve. It is more learnt to the problem domain and if services splits or merges, the end-to-end can still tests that the business functions are intact after changes. For a distributed system on the stage of LiMiC1.0, that is very likely to more functionality around, this is ideal (Note: a unit test can be good when developing and testing relative small pieces of code. End-to-end test can be part of the cluster-startup sequence to make sure everything responds and works). This also rely on the fact, that the LiMiC System's services is maintained by the team itself [33].

6.3 Running Tests

Startup the docker container and pipe a command to the container using `exec`. The tests is are written to be running on a initially empty database. As long as the drones is defined with manually types ids (Waiting for the VRP and UI to rely on Drone Service for drone ids), this is preferable, as the other services should'nt create the drones before calling the services either. Make sure that the `core/config.py` is updated in each service with the correct IP/URL for the integrating services.

Docker Compose

```
$ docker-compose up -d --build
$ docker-compose exec app pytest .
```

Figure 6.1: .Get request called on the TestClient

Docker (Single Container)

```
$ docker build -t estimators .
```

Figure 6.2: Building the Estimator Service from its Dockerfile

Build:
The `-t` is a tag, that makes it possible to call the built image on its name when starting (instead of generated container id.)

Run Image - Creating Container:

```
$ docker run -p 8080:8080 estimators
```

Listing 2: .Get request called on the TestClient

```
$ docker run --network=logs_default -p 8080:8080 estimator
```

Figure 6.3: .Get request called on the TestClient

Run Tests:

```
$ docker exec -it <CONTAINER ID>
$ pytest .
or
$ pytest . --cov
```

Listing 3: .Get request called on the TestClient

Test Results

The test written can be seen in the test directories. They tests, tests each endpoint directly and is formed as integration-tests. The Estimator Service has a code coverage on 89%. See the specification for each file in appendix 9.6.1. The Drone/Logger Service test 9.7 has a test coverage on 87% and the Mission Service has a test coverage on 88% The test-result and coverage rapport can be seen in appendix. It would be see that almost all of the business logic is tested an the code that is not running in every service is primarily code in application core.

Chapter 7

Evaluation

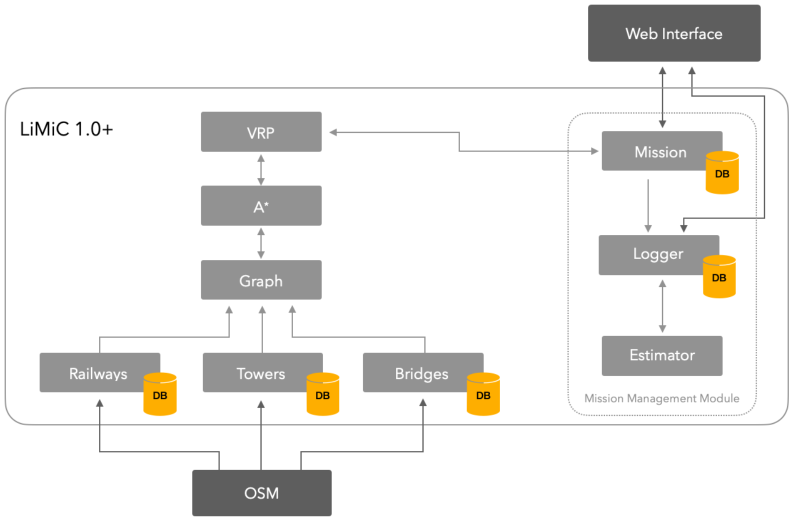


Figure 7.1: Microservice Architecture of LiMiC1.0+

LiMiC1.0+ consist of three services, that meets the requirements for the current stage of the project described in Drone4Safety Specification [29] and integrate with the existing other services and deployment system described in *From Monolith to Microservice: Software Architecture for Autonomous UAV Infrastructure Inspection* [28].

The microservices is implemented using a 3-layer architecture, splitting the areas of concern. This gives a good functionality seperation, making each layer easy to maintain and extend. The respective layers: CL, BLL, DAL heavily relies on the same building blocks and concepts, with very few application or internal service changes. This makes makes it very easy to read and understand. Handling of database connection is build in to the objects, and creating a new service that relies on a database only requires to inherit the `AppService` class. This is thanks to the application `core` making a platform for internal services, both with and without the use of DB. The core constructs and starts up the application. If database abstraction is build in, the core creates the connection. If anythings goes wrong during the connection, a error message is printed to stdout.
The database abstraction, thanks to SQLAlchemy and GeoAlchemy2 makes a easy integration into Python. Further they makes is easy to changes RDBMS if needed. Here the reader should be aware, that not many other RDBMS supports all the functionality implemented here.
Thanks to Uvicorn, FastAPI and Pydantic, the applications is quite fast. The use of these three dependencies in the implementation makes a easy to create a fast and reliable API. Pydantic validates everything going in and out of the application. FastAPI abstracts upon Starlette, almost hiding protocol communication. Referring the section about selecting the RDBMS, PostgreSQL with PostGIS extension serves as the best solution to the requirements for the use case. Both taking the types and geospatial functionalities and the data integrity into account, that is important when controlling and monitoring UAVs. All in all a tech stack that fits the requirements well used to

develop a platform for services, that is very extendable and flexible as the project grows or changes.

The reader might critic that the CRUD implementation do not utilize the generic possibilities within the internal services. This is a suggested structure in the example application of a full stack FastAPI application (Created by FastAPI Author) [60]. Because we use special geo-spatial types, it would require to inherit the base an overwrite a lot of generic functions. This would make a more complex application, with lower readability. With the chosen strategy the business logic and crud is in the same service file for each service (using external schemas dedicated to each service). A arbitrary router can then call the each function on a constructed class. All with BLL and CRUD error handling. This is what justifies the use of a BLL layer. In many of the internal services it could seems redundant to the essential CRUD. The structure encapsulate each service better, while together while having a reasonable degree of cohesion between the services.

The estimator is implemented using vectors, affecting each dimension separately. This is a lightweight computation (In difference from using trigonometry). The estimator works, but only within a certain time-slot. Using the estimator on telemetry-data that is hours old, would not be valid. This would probably course a error from the Estimator, as Pydantic creates an error message, when constructing non-valid schemas.
The solutions handles the asynchronous / synchronous cases as suggested by both FastAPI and SQLAlchemy, making the best performant solution to this use case.

There could be added many more tests, testing different response codes. The tests written is primarily to make sure each endpoints answers and works.

7.0.1 Future Optimizations and Changes

Alembic

When the databases start to contain data that should be persistent, maybe for further development but in any case before production, Alembic should be implemented. It is a lightweight database migration tool for use with SQLAlchemy. It is written by SQLAlchemy's author, and eases the migration to updated database relations.

Background tasks

Creating a route is immediately giving the client a response with the id, to create a mission with the given route. This means the client need to implement something like a exponential backoff algorithm to create the mission, because the fetching of the route could fail (if the VRP doesn't have returned yet). The implementation is requested to be like this for now.
A more ideal implementation would be with a notification send upon route computation completed and persisted. This can be implemented using websocket. Websockets keeps the connection alive after the HTTP (Hyper Text Transfer Protocol) response is received, facilitates a bi-directional message passing between client a server. This is done over HTTP through a single TCP/IP socket connection [26]. Websocket is supported by FastAPI [73].

Data Hiding and Ids

Currently the requests are exposing the raw database ids (Primary Keys). Exposing PK's to a database in production could expose a security risk (but is convenient in development). At some point, when the applications is mature enough, a strategy to prevent a attack utilizing this should be considered. Hashing could be a strategy for this. Another alternative is to look towards e.g. Instagram [24], that using a sharding strategy to distribute data between any number of PostgreSQL Servers. The sharding id, time and a autoincrementing sequence is the part of generating unique ids across databases (logically mapped in code to fewer shards). Using the time as part of the id, makes sorting faster (which is not a strong feature with UUID). Utilizing sharding could be another solution to the suggested Lambda architecture migration mentioned in *Specification of the Drone Inspection as a Service platform* [29].

Estimator

Currently the estimator returns the estimated position. This is on request. Another solution, that could course in better visualizing (Note: not necessary more precise prediction of the actual position), is returning the "speed"-vector computed. With the vector the UI could continuously update without calling the Estimator in between

reporting-cycle from the drone.
When the Whether Service and the specification on the drones Is known, the estimator can be extended with the estimated drone-status.

Chapter 8

Thesis Conclusion

Through this thesis we have discovered the need for a cloud service in the D4S project, fostering the need of LiMiC. The existing literature, specifications and research papers are explored and incorporated into the thesis, making a basis for the reason of developing the three microservices designed and implemented throughout this thesis. The existing code base is explored. Getting to know the different libraries and other dependencies already used in the project. The existing services and their functionality is studied to getting to know their logic and how to integrate with them, to extend the overall LiMiC1.0 system to LiMiC1.0+.
We have designed two relation databases serving as persistent storage virtual assets. The databases meets the basic requirements for good database design. Missions can be planned, together with a swam of drones. The drones request their specific flight plans and they can report back detected faults during their inspection. The drones position and status is tracked using the telemetries reported back. Further drones positions is estimated using the reported geo-locations using the third estimator service. The Missions services integrate with the existing VRP service, to get routes computed, as this is now requested through this service.
All in all is 3 microservices consisting of 9 internal services is delivered. The thesis meets all the requirements given in the problem statement, together with the changes and additions added through project period.

Chapter 9

Appendix

9.1 Docker Compose Startup

This is an example from starting up the Missions Service. It is identical with the Drone/Logger Service.

```
Jacobs-MacBook-Pro-2:logs jacobnielsen$ docker compose up
[+] Running 2/2
Container logs_db_1   Created 0.0s
Container logs_app_1  Recreated 0.2s
Attaching to app_1, db_1
```

Figure 9.1: Docker Command

```
    db_1    |
    db_1    | PostgreSQL Database directory appears to contain a database; Skipping
↪   initialization
    db_1    |
    db_1    | 2021-12-23 15:55:00.603 UTC [1] LOG:  starting PostgreSQL 13.4 on
↪   x86_64-pc-linux-musl,
                        compiled by gcc (Alpine 10\.\3.1\_git20210424)10\.3\.1 20210424, 64-bit
    db_1    | 2021-12-23 15:55:00.603 UTC [1] LOG:  listening on IPv4 address "0.0.0.0", port
↪   5432
    db_1    | 2021-12-23 15:55:00.603 UTC [1] LOG:  listening on IPv6 address "::", port 5432
    db_1    | 2021-12-23 15:55:00.607 UTC [1] LOG:  listening on Unix socket
                        "/var/run/postgresql/.s.PGSQL.5432"
    db_1    | 2021-12-23 15:55:00.617 UTC [13] LOG:  database system was shut down at
                        2021-12-21 22:45:31 UTC
    db_1    | 2021-12-23 15:55:00.627 UTC [1] LOG:  database system is ready to accept
↪   connections
```

Figure 9.2: Database startup

```
app_1   | INFO:     Started server process [1]
app_1   | INFO:     Waiting for application startup.
```

Figure 9.3: Uvicorn webserver startup

```
    app_1   | 2021-12-23 15:55:03,985 INFO sqlalchemy.engine.Engine select pg_catalog.version()
    app_1   | 2021-12-23 15:55:03,985 INFO sqlalchemy.engine.Engine [raw sql] {}
    app_1   | 2021-12-23 15:55:03,990 INFO sqlalchemy.engine.Engine select current_schema()
    app_1   | 2021-12-23 15:55:03,990 INFO sqlalchemy.engine.Engine [raw sql] {}
    app_1   | 2021-12-23 15:55:03,994 INFO sqlalchemy.engine.Engine show
↪   standard_conforming_strings
    app_1   | 2021-12-23 15:55:03,994 INFO sqlalchemy.engine.Engine [raw sql] {}
    app_1   | 2021-12-23 15:55:03,996 INFO sqlalchemy.engine.Engine BEGIN (implicit)
    app_1   | 2021-12-23 15:55:03,997 INFO sqlalchemy.engine.Engine
            select relname from pg_class c join pg_namespace n on n.oid=c.relnamespacewhere
                pg_catalog.pg_table_is_visible(c.oid) and relname=%(name)s
    app_1   | 2021-12-23 15:55:03,997 INFO sqlalchemy.engine.Engine
                        [generated in 0.00045s] {'name': 'Telemetry'}
    app_1   | 2021-12-23 15:55:04,004 INFO sqlalchemy.engine.Engine
            select relname from pg_class c join pg_namespace n on n.oid=c.relnamespacewhere
                pg_catalog.pg_table_is_visible(c.oid) and relname=%(name)s
    app_1   | 2021-12-23 15:55:04,005 INFO sqlalchemy.engine.Engine [cached since 0.007884s ago]
            {'name': 'drone'}
    app_1   | 2021-12-23 15:55:04,007 INFO sqlalchemy.engine.Engine select relname from pg_class
↪   c
            join pg_namespace n on n.oid=c.relnamespacewhere
↪   pg_catalog.pg_table_is_visible(c.oid)
                and relname=%(name)s
    app_1   | 2021-12-23 15:55:04,007 INFO sqlalchemy.engine.Engine [cached since 0.0103s ago]
            {'name': 'swam'}
    app_1   | 2021-12-23 15:55:04,010 INFO sqlalchemy.engine.Engine select relname from pg_class
↪   c
            join pg_namespace n on n.oid=c.relnamespacewhere
↪   pg_catalog.pg_table_is_visible(c.oid)
                and relname=%(name)s
    app_1   | 2021-12-23 15:55:04,010 INFO sqlalchemy.engine.Engine [cached since 0.01331s ago]
            {'name': 'estimates'}
    app_1   | 2021-12-23 15:55:04,012 INFO sqlalchemy.engine.Engine COMMIT
    app_1   | INFO:     Application startup complete.
    app_1   | INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
```

Figure 9.4: Application startup

9.2 Docker Compose Shutdown

9.3 Models

Route

```
            class Route(Base):
                __tablename__ = 'route'
                rId = Column(UUID(as_uuid=True), primary_key=True) #, default=uuid4) # we need
                 ↪  to have it
                route = Column(JSONB)
```

Listing 4: Response Handler for internal service response

```
Gracefully stopping... (press Ctrl+C again to force)
[+] Running 2/2
Container logs_app_1 Stopped 0.5s
Container logs_db_1   Stopped 0.2s
canceled
```

Figure 9.5: Docker compose shut down

Mission

```
            class Mission(Base):
                __tablename__ = 'missions'
                mId = Column(UUID(as_uuid=True), primary_key=True)
                name = Column(String)
                done = Column(Boolean, default=False)
                active = Column(Boolean, default=False)
                swamId = Column(UUID(as_uuid=True)) # also called alloc
                routeId = Column(UUID(as_uuid=True)) #
                time_created = Column(DateTime, default=datetime.datetime.utcnow)
                time_updated = Column(DateTime, server_default=func.now(),
                 ↪  server_onupdate=func.now())

                # Relationships
                tasks = relationship('Task',  backref='mission')  # one-to-many relation to
                 ↪  tasks
                results = relationship('Result', backref="mission") # one-to-many relation to
                 ↪  results
```

Task

```
        class Task(Base):
            __tablename__ = 'tasks'
            tId = Column(UUID(as_uuid=True), primary_key=True)                  # id
            iType = Column(String)                                              # inspection
             ↪  type
            loc = Column(Geometry('POINTZ', dimension=3))                       # location
            dSpec = Column(String)                                              # data
             ↪  specification
            dId = Column(String)                                                # drone id
            # Relationships
            missionId = Column(UUID(as_uuid=True), ForeignKey('missions.mId'))  # foregin key
```

Telemtry

```
        class Telemetry(Base):
            __tablename__ = 'Telemetry'
            lId = Column(UUID(as_uuid=True), primary_key=True)                      # log ids
            mId = Column(UUID(as_uuid=True))                                        #
             ↪  missionId
            time_created = Column(DateTime, default=datetime.datetime.utcnow)
            ori = Column(ARRAY(Float))                                              #
             ↪  orientation of drone roll, pitch, and yaw angles of the drone.
            pos = Column(Geometry('POINTZ'))                                        # position
             ↪  GNSS
            vel = Column(Float)                                                     #
             ↪  Velocity
        sta = Column(String)                                         # Flight
        ↪  Status
        link = Column(Integer)                                       #
        ↪  connection strenght
        batt = Column(Integer)                                       # battery
        ↪  status
        cpu = Column(Integer)                                        # CPU
        ↪  status
        mem = Column(Integer)                                        # Memory
        ↪  status
        # Relationships
        dId = Column(String, ForeignKey('drone.dId'))                 # foregin
        ↪  key,   # drone ID
```

Drone

```
    class Drone(Base):
        __tablename__ = 'drone'
        dId = Column(String, primary_key=True)
        # Relationships
        # One to Many
        estimations = relationship('Estimate',  backref='estimates')
        # bi-directional
        swams = relationship('Swam', backref="drones")
        telemetries = relationship('Telemetry', backref="drone")
```

Result

```
    class Result(Base):
        __tablename__ = 'results'
        rId = Column(UUID(as_uuid=True), primary_key=True)              # "If you want
        ↪  to pass a UUID() object, the as_uuid flag must be set to True."
        time_created =  Column(DateTime, default=datetime.datetime.utcnow) # time
        loc = Column(Geometry('POINTZ', dimension=3, srid=4326))          # location
        fau = Column(String)                                             # fault
        ↪  component type / fault type
        img = Column(UUID(as_uuid=True))                                 # image id
        dId = Column(String)                                             # drone id
        # Relationships
        missionId = Column(UUID(as_uuid=True), ForeignKey('missions.mId')) # foregin key
```

Swam

```
    class Swam(Base):
        __tablename__ = 'swam'
        rowId = Column(Integer, primary_key=True)                  # Should auto
        ↪  increment
        sId = Column(UUID(as_uuid=True))                           # Shared swam id
        # Relationships
        dId = Column(String, ForeignKey('drone.dId'))              # drone id
```

Estimate

```
    class Estimate(Base):
        __tablename__ = 'estimates'
```

```
            eId = Column(UUID(as_uuid=True), primary_key=True)
            mId = Column(UUID(as_uuid=True))                                     #
            ↪   missionId
            time_created = Column(DateTime, default=datetime.datetime.utcnow)
            ePos = Column(Geometry('POINTZ'))                                    #
            ↪   estimated position
            # Relationships
            dId = Column(String, ForeignKey('drone.dId'))                        # foregin
            ↪   key to drones
```

9.4 Shapely versus Geoalchemy2

Testing the performance-difference between using Shapely to extract data from WBK object and using PostGIS Spatial function (With GeoAlchemy2 abstraction). The test is performed by inserting single Telemetry entity into the database and make continuos queries on this object, with SELECT operation; only reading and hence no locking by DBMS.
The test is performed using a Client script. The test is performed in a local environment and to not take network round trip times, bandwidth or congestion into account. Each test is running in synchronous way before the same test run with the requests in a thread pool
The implementation with Shapely and the conversion done in Logic Layer is performing better in every case.

Client Script:

```
import functools
import time
import requests
from concurrent.futures import ThreadPoolExecutor as Pool


tele_post ={
            "mId": "3fa85f64-5717-4562-b3fc-2c963f66afa7",
            "dId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
            "pos": {
                "type": "Point",
                "coordinates": {
                "lat": 10.42078971862793,
                "lng": 55.36930763445593,
                "hei": 10
                }
            },
            "vel": 9.7,
            "sta": "string",
            "link": 80,
            "batt": 76,
            "cpu": 56,
            "mem": 34
            }

def timed(N, url, fn):
    @functools.wraps(fn)
    def wrapper(*args, **kwargs):
        start = time.time()
        res = fn(*args, **kwargs)
        stop = time.time()
        duration = stop - start
        print(f"{N / duration:.2f} reqs / sec | {N} reqs | {url} | {fn.__name__}")
```

```python
        return res

    return wrapper

def get(url):
    resp = requests.get(url)

    assert resp.status_code == 200
    return resp.json()


def sync_get_all(url, n):
    l = [get(url) for _ in range(n)]
    return l

def run_bench(n, funcs, urls):
    for url in urls:
        for func in funcs:
            timed(n, url, func)(url, n)

def post_tele():
    for _ in range(10):
        response = requests.post(url = "http://127.0.0.1:8008/telemetry", json = tele_post)
        res = response.content.decode('utf-8')
        print(res)


# multiple simultaneous connections:
def thread_pool(url, n, limit=None):
    limit_ = limit or n
    with Pool(max_workers=limit_) as pool:
        result = pool.map(get, [url] * n)
    return result


if __name__ == "__main__":
    # post teles
    post_tele()

    # get
    urls = ["http://127.0.0.1:8008/telemetry/3fa85f64-5717-4562-b3fc-2c963f66afa7"]
    funcs = [sync_get_all]
    print("running bench mark")

    run_bench(500, [sync_get_all, thread_pool], urls)
```

Results:

```
# WITHOUT SHAPELY
# 31.45 reqs / sec | 100 reqs |
↪  http://127.0.0.1:8008/telemetry/3fa85f64-5717-4562-b3fc-2c963f66afa7 | sync_get_all
# 28.74 reqs / sec | 100 reqs |
↪  http://127.0.0.1:8008/telemetry/3fa85f64-5717-4562-b3fc-2c963f66afa7 | thread_pool

# WITH SHAPELY
# running bench mark
# 62.23 reqs / sec | 100 reqs |
↪  http://127.0.0.1:8008/telemetry/3fa85f64-5717-4562-b3fc-2c963f66afa7 | sync_get_all
# 35.46 reqs / sec | 100 reqs |
↪  http://127.0.0.1:8008/telemetry/3fa85f64-5717-4562-b3fc-2c963f66afa7 | thread_pool

------------------------------------------------------------------------------------------
# WITHOUT SHAPELY
# running bench mark
# 41.35 reqs / sec | 200 reqs |
↪  http://127.0.0.1:8008/telemetry/3fa85f64-5717-4562-b3fc-2c963f66afa7 | sync_get_all
# 53.00 reqs / sec | 200 reqs |
↪  http://127.0.0.1:8008/telemetry/3fa85f64-5717-4562-b3fc-2c963f66afa7 | thread_pool

# WITH SHAPELY
# 63.28 reqs / sec | 200 reqs |
↪  http://127.0.0.1:8008/telemetry/3fa85f64-5717-4562-b3fc-2c963f66afa7 | sync_get_all
# 38.68 reqs / sec | 200 reqs |
↪  http://127.0.0.1:8008/telemetry/3fa85f64-5717-4562-b3fc-2c963f66afa7 | thread_pool


------------------------------------------------------------------------------------------
# WITHOUT SHAPELY
# 47.43 reqs / sec | 300 reqs |
↪  http://127.0.0.1:8008/telemetry/3fa85f64-5717-4562-b3fc-2c963f66afa7 | sync_get_all
# 4.54 reqs / sec | 300 reqs |
↪  http://127.0.0.1:8008/telemetry/3fa85f64-5717-4562-b3fc-2c963f66afa7 | thread_pool

# WITH SHAPELY
# 48.04 reqs / sec | 300 reqs |
↪  http://127.0.0.1:8008/telemetry/3fa85f64-5717-4562-b3fc-2c963f66afa7 | sync_get_all
# 8.04 reqs / sec | 300 reqs |
↪  http://127.0.0.1:8008/telemetry/3fa85f64-5717-4562-b3fc-2c963f66afa7 | thread_pool
```

9.5 TestClient

This is an example on how the `TestClient` can be used like like `request`

```python
def test_read_main():
    response = client.get("/")
    assert response.status_code == 404
    assert response.json() == {"detail":"Not Found"}
```

Figure 9.6: .Get request called on the TestClient

Drone/Logger Service

9.6 Estimator Tests

This as a complete example for using the Estimator. Most confusing to new users is probably, that it doesn't use time-zone extensions - Hence running on UTC-0 like the databases in the two other microservices.

Constant Speed Evaluation

This evaluation is forces by only give 2 positions to the Estimator. As the estimator service is calculation the position using "Now" the request is send with approximately 30s after the time on the newest position.

Request:

```
    [
        {
          "time_created": "2021-12-29T09:11:30.00",
          "point": {
            "type": "Point",
            "coordinates": {
              "lat": 10.416905879974363,
              "lng": 55.37016120977823,
              "hei": 10.0
            }
          }
        },
      {
          "time_created": "2021-12-29T09:11:00.00",
          "point": {
            "type": "Point",
            "coordinates": {
              "lat": 10.418579578399658,
              "lng": 55.36938079877647,
              "hei": 10.0
            }
          }
        }
      ]
```

Listing 5: Request Body simulating two Drone positions

Response (Body):

```
{
    "point": {
        "type": "Point",
        "coordinates": {
        "lat": 10.419979409105302,
        "lng": 55.36872808663503,
        "hei": 10
        }
    }
}
```

Listing 6: Request Body simulating two Drone positions

Here the benefits of the use of vectors is shown. The Z-coordinate (hei) does'nt change as there is only change in the other two dimensions.

Map Plot:

Constant Acceleration Speed Estimation:

This evaluation is forces by only give 3 positions to estimator, where the distance between the second and third point is bigger (minimum 5%) than between first and second.

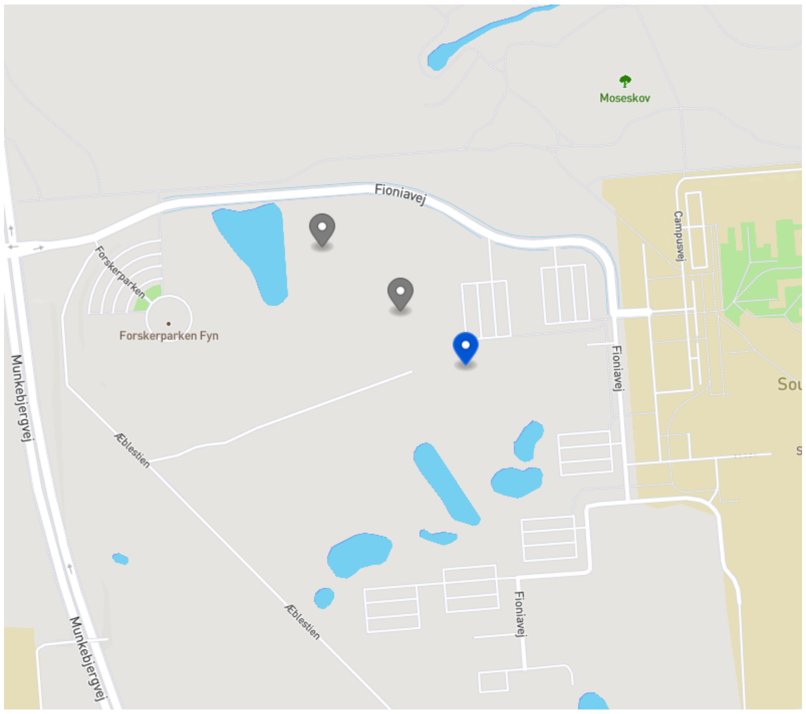


Figure 9.7: [Constant Speed] Estimation Point (blue) - Internal Services

Request:

```
        [
            {
              "time_created": "2021-12-29T14:19:00.00",
              "point": {
                "type": "Point",
                "coordinates": {
                  "lat": 10.416905879974363,
                  "lng": 55.37016120977823,
                  "hei": 10.0
                }
              }
            },
          {
              "time_created": "2021-12-29T14:18:00.00",
              "point": {
                "type": "Point",
                "coordinates": {
                  "lat": 10.418579578399658,
                  "lng": 55.36938079877647,
                  "hei": 10.0
                }
              }
            },
          {
              "time_created": "2021-12-29T14:17:00.00",
              "point": {
                "type": "Point",
                "coordinates": {
                    "lat": 10.422060363324169,
                    "lng": 55.3677577806695,
                    "hei": 10.0
                }
            }
        }
    ]
```

Listing 7: Request Body simulating two Drone positions

Response:

```
    {
        "point": {
            "type": "Point",
            "coordinates": {
            "lat": 10.427619512195722,
            "lng": 55.36516566438724,
            "hei": 10
            }
        }
    }
```

Listing 8: Request Body simulating two Drone positions

Map Plot: Estimations Point (Blue) for Constant Acceleration Estimation. As for Constant evaluation the current time is taken into account. There is approximately 30s from the last of the grey points, to the estimation is requested. Overall Constant Acceleration is generating quite big jumps.

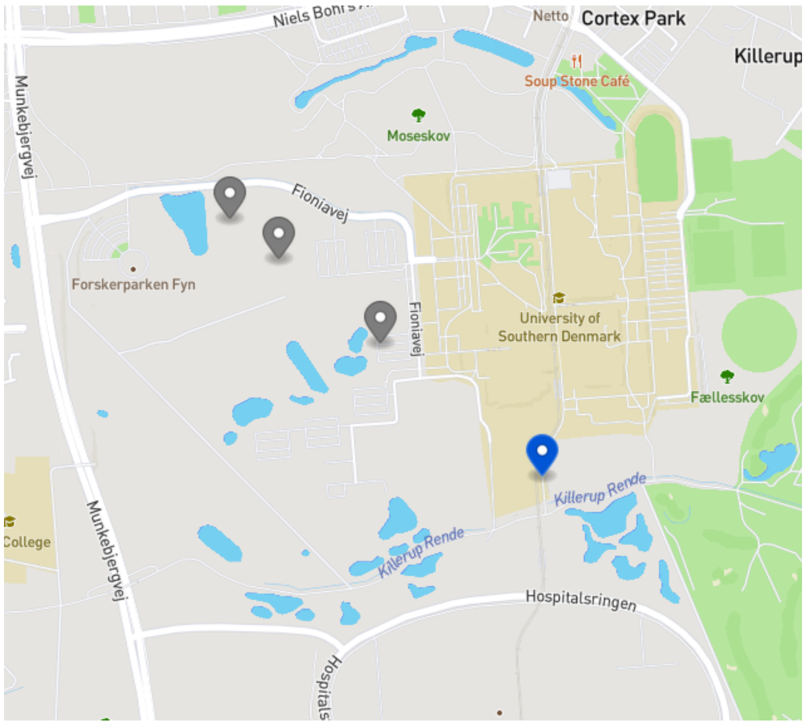


Figure 9.8: [Constant Acceleration] Estimation Point (blue) - Internal Services

9.6.1 Running Pytest

```
============================= test session starts ==============================
platform linux -- Python 3.9.9, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /estimator
plugins: anyio-3.4.0, cov-3.0.0, asyncio-0.16.0
collected 4 items

tests/test_main.py ....                                                  [100%]

----------- coverage: platform linux, python 3.9.9-final-0 -----------
Name                                  Stmts   Miss  Cover
---------------------------------------------------------
app/__init__.py                           0      0   100%
app/core/__init__.py                      0      0   100%
app/core/config.py                        8      0   100%
app/core/exception.py                    28     13    54%
app/core/response.py                     30     11    63%
app/main.py                              11      1    91%
app/parsers/__init__.py                   0      0   100%
app/parsers/parser.py                    14      0   100%
app/routers/__init__.py                   0      0   100%
app/routers/router.py                    10      0   100%
app/schemas/__init__.py                   0      0   100%
app/schemas/geometries.py                27      4    85%
app/schemas/schemas.py                    8      0   100%
app/services/__init__.py                  0      0   100%
app/services/estimator_service.py        85      4    95%
app/utils/__init__.py                     0      0   100%
app/utils/estimator.py                   29      0   100%
app/utils/vectorposition.py              15      1    93%
tests/__init__.py                         0      0   100%
tests/test_main.py                       33      0   100%
---------------------------------------------------------
TOTAL                                   298     34    89%

============================== 4 passed in 0.63s ===============================
```

Listing 9: Pytest with coverage Rapport

9.7 Drone/Logger Tests

9.7.1 Running Pytest

```
================================================================================ test
↪  session starts
↪  ================================================================================
platform linux -- Python 3.9.9, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /logs
plugins: anyio-3.4.0, cov-3.0.0, asyncio-0.16.0
collected 19 items

tests/test_drone.py ......
↪  [ 31%]
tests/test_estimate.py ..
↪  [ 42%]
```

```
tests/test_swam.py .....
↪  [ 68%]
tests/test_telemetry.py ......
↪  [100%]

=================================================================================
↪  warnings summary
↪  ================================================================================
tests/test_estimate.py::TestEstimator::test_estimator_no_acceleration
tests/test_estimate.py::TestEstimator::test_estimator_with_acceleration
  /logs/app/services/telemetry_service.py:209: SAWarning: UserDefinedType
  ↪  Geometry(geometry_type='POINTZ', from_text='ST_GeomFromEWKT', name='geometry')
  ↪  will not produce a cache key because the ``cache_ok`` attribute is not set to
  ↪  True.  This can have significant performance implications including some
  ↪  performance degradations in comparison to prior SQLAlchemy versions.  Set this
  ↪  attribute to True if this type object's state is safe to use in a cache key, or
  ↪  False to disable this warning. (Background on this error at:
  ↪  https://sqlalche.me/e/14/cprf)
    db_teles = self.db.query(

tests/test_telemetry.py::TestTelemetry::test_get_path_by_drone_id
  /logs/app/services/telemetry_service.py:200: SAWarning: UserDefinedType
  ↪  Geometry(geometry_type='POINTZ', from_text='ST_GeomFromEWKT', name='geometry')
  ↪  will not produce a cache key because the ``cache_ok`` attribute is not set to
  ↪  True.  This can have significant performance implications including some
  ↪  performance degradations in comparison to prior SQLAlchemy versions.  Set this
  ↪  attribute to True if this type object's state is safe to use in a cache key, or
  ↪  False to disable this warning. (Background on this error at:
  ↪  https://sqlalche.me/e/14/cprf)
    db_points = self.db.query(Telemetry.pos).filter(Telemetry.dId == dId_in).all() #
    ↪  without .all() you have a query object

-- Docs: https://docs.pytest.org/en/stable/warnings.html

----------- coverage: platform linux, python 3.9.9-final-0 -----------
Name                                Stmts   Miss  Cover
-------------------------------------------------------
app/core/__init__.py                    0      0   100%
app/core/config.py                     25      2    92%
app/core/exception.py                  28     13    54%
app/core/response.py                   26      8    69%
app/core/tasks.py                      13      4    69%
app/db/__init__.py                      0      0   100%
app/db/database.py                      8      0   100%
app/db/sessionconnect.py                6      4    33%
app/db/tasks.py                        32     17    47%
app/main.py                            20      1    95%
app/models/__init__.py                  0      0   100%
app/models/drone.py                    11      0   100%
app/models/estimate.py                 14      0   100%
app/models/swam.py                      8      0   100%
app/models/telemetry.py                20      0   100%
app/routers/__init__.py                 0      0   100%
app/routers/router_drone.py            31      3    90%
app/routers/router_estimate.py         18      3    83%
app/routers/router_swam.py             28      1    96%
```

```
app/routers/router_telemetry.py       41      7    83%
app/schemas/__init__.py                0      0   100%
app/schemas/drone.py                  14      0   100%
app/schemas/estimate.py               19      0   100%
app/schemas/swam.py                   12      0   100%
app/schemas/telemetry.py              45      0   100%
app/schemas/utils/geometries.py       48      3    94%
app/schemas/utils/positioning.py       5      0   100%
app/services/__init__.py               0      0   100%
app/services/drone_service.py         76     15    80%
app/services/estimator_service.py     86     17    80%
app/services/serviceconnect.py         8      0   100%
app/services/swam_service.py          63      4    94%
app/services/telemetry_service.py    107     15    86%
app/utils/geometryConverter.py        22      3    86%
tests/__init__.py                      0      0   100%
tests/test_drone.py                   30      0   100%
tests/test_estimate.py                33      0   100%
tests/test_main.py                    16      0   100%
tests/test_swam.py                    61      0   100%
tests/test_telemetry.py               89     13    85%
-------------------------------------------------------
TOTAL                               1063    133    87%

============================================================================ 19 passed,
↪  3 warnings in 3.86s
↪  ===========================================================================
```

Listing 10: Pytest with coverage Rapport

The 2 warnings is generated because that `cache_ok` (allowing or not allowing SQLAlchemy to cache the type) is not set in the defined types within GeoAlchemy2. Hence this should be updated within that dependency's types, that we implement.

9.8 Missions Tests

9.8.1 Running Pytest

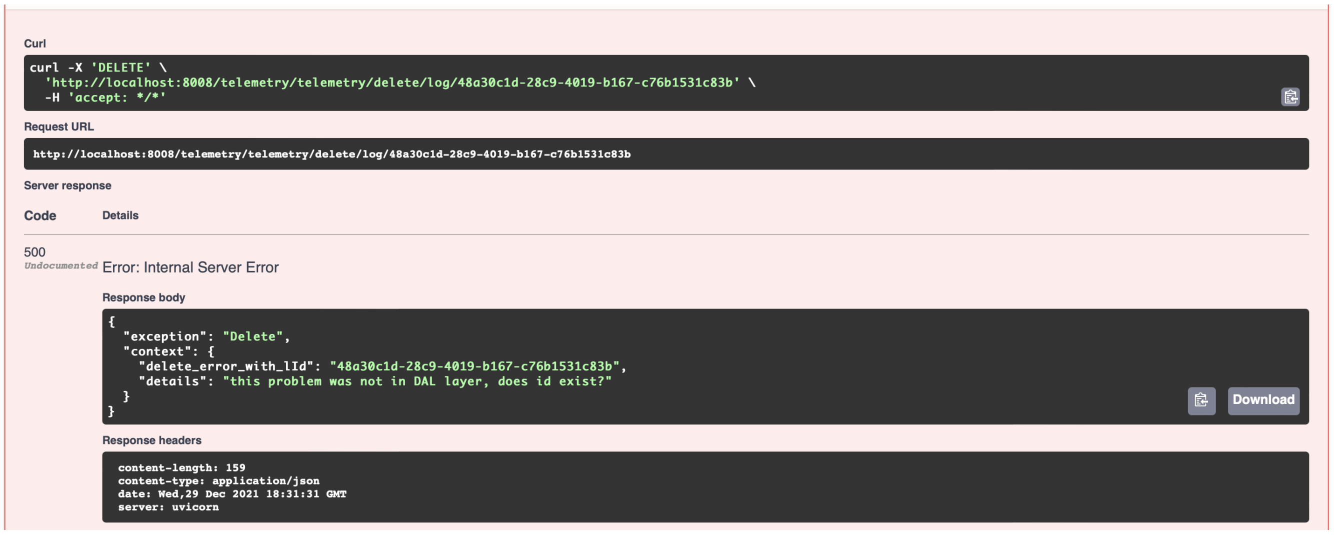
```
==============================================================================
↪  test session starts
↪  ===============================================================================
platform linux -- Python 3.9.9, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /missions
plugins: anyio-3.4.0, cov-3.0.0, asyncio-0.16.0
collected 16 items
tests/test_mission.py ......
↪  [ 37%]
tests/test_result.py .....
↪  [ 68%]
tests/test_route.py ...
↪  [ 87%]
tests/test_task.py ..
↪  [100%]

----------- coverage: platform linux, python 3.9.9-final-0 -----------
Name                              Stmts   Miss  Cover
```

```
-----------------------------------------------------
app/core/__init__.py                   0      0   100%
app/core/config.py                    32      2    94%
app/core/exception.py                 29     13    55%
app/core/response.py                  30      9    70%
app/core/tasks.py                     13      4    69%
app/db/__init__.py                     0      0   100%
app/db/database.py                    10      0   100%
app/db/sessionconnect.py               6      4    33%
app/db/tasks.py                       32     17    47%
app/main.py                           20      1    95%
app/models/__init__.py                 0      0   100%
app/models/mission.py                 20      0   100%
app/models/result.py                  16      0   100%
app/models/route.py                    8      0   100%
app/models/task.py                    13      0   100%
app/parsers/__init__.py                0      0   100%
app/parsers/vrp_parser.py             22      0   100%
app/routers/__init__.py                0      0   100%
app/routers/router_missions.py        41      5    88%
app/routers/router_results.py         30      1    97%
app/routers/router_router.py          29      6    79%
app/routers/router_tasks.py           27      5    81%
app/schemas/__init__.py                0      0   100%
app/schemas/mission.py                40      1    98%
app/schemas/result.py                 18      0   100%
app/schemas/route.py                  38      1    97%
app/schemas/swam.py                    6      0   100%
app/schemas/task.py                   22      0   100%
app/schemas/utils/__init__.py          0      0   100%
app/schemas/utils/geometries.py       12      1    92%
app/services/mission_service.py      154     20    87%
app/services/result_service.py        74      5    93%
app/services/route_service.py         56      4    93%
app/services/serviceconnect.py         8      0   100%
app/services/task_service.py          83     26    69%
app/utils/geometryConverter.py        21      3    86%
tests/__init__.py                      0      0   100%
tests/test_main.py                    17      0   100%
tests/test_mission.py                 70      0   100%
tests/test_result.py                  82      4    95%
tests/test_route.py                   27      0   100%
tests/test_task.py                    24      0   100%
-----------------------------------------------------
TOTAL                               1130    132    88%


================================================================================== 16
↪  passed in 12.50s
↪  ==================================================================================
```

Listing 11: Pytest with coverage Rapport

9.9 Custom Exception in Action

This implementation does do a great deal with error handling, giving the client helpful error-messages, while still keeping the crucial parts (from DAL and SQLAlchemy) within. When deleting on a PK that doesn't exist in DB, here specifically deleting a Telemetry entry in its `lId`, the response would be the following (on Docs page.) Everything in the context dictionary are parsed directly from within the lines of the code, making it super easy to parse the detail you want through the system to the client.



Curl

```
curl -X 'DELETE' \
  'http://localhost:8008/telemetry/telemetry/delete/log/48a30c1d-28c9-4019-b167-c76b1531c83b' \
  -H 'accept: */*'
```

Request URL

```
http://localhost:8008/telemetry/telemetry/delete/log/48a30c1d-28c9-4019-b167-c76b1531c83b
```

Server response

Code Details

500 *Undocumented* Error: Internal Server Error

Response body

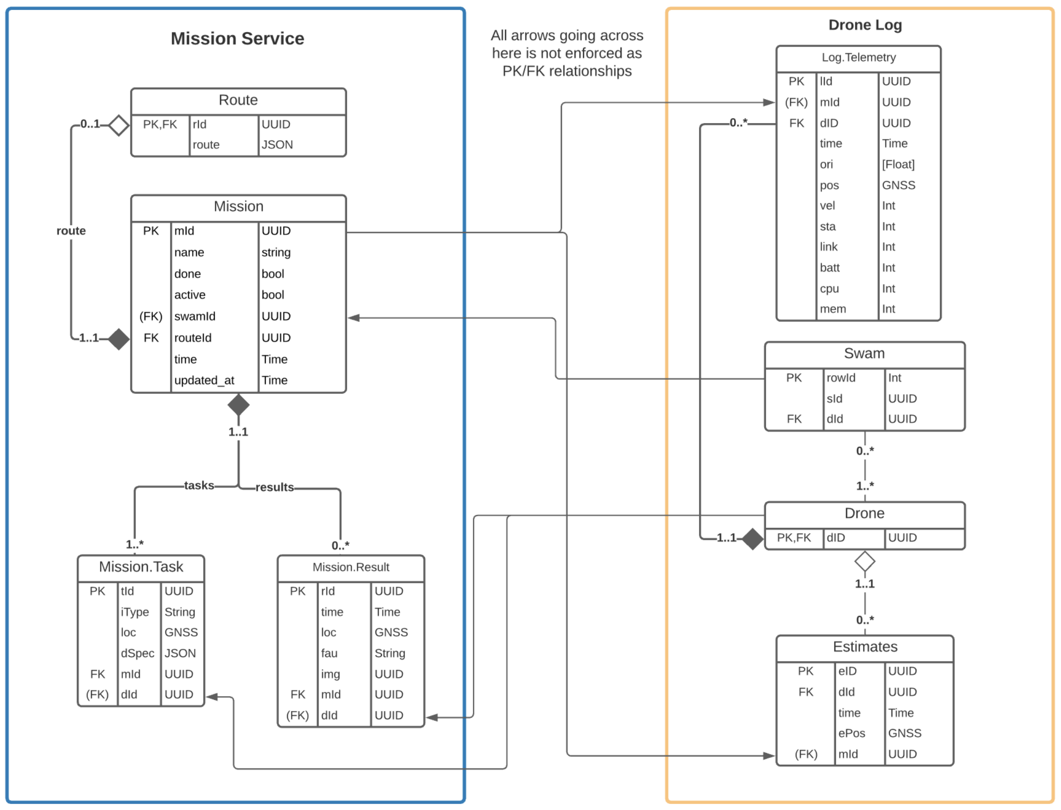
```
{
  "exception": "Delete",
  "context": {
    "delete_error_with_lId": "48a30c1d-28c9-4019-b167-c76b1531c83b",
    "details": "this problem was not in DAL layer, does id exist?"
  }
}
```

Response headers

```
content-length: 159
content-type: application/json
date: Wed,29 Dec 2021 18:31:31 GMT
server: uvicorn
```

Figure 9.9: Custom Exception Response

9.10 Data Structure - how they keys bind the system together



9.11 Dependencies

Below here is all the dependencies from the `requirement.txt` files

Without Database

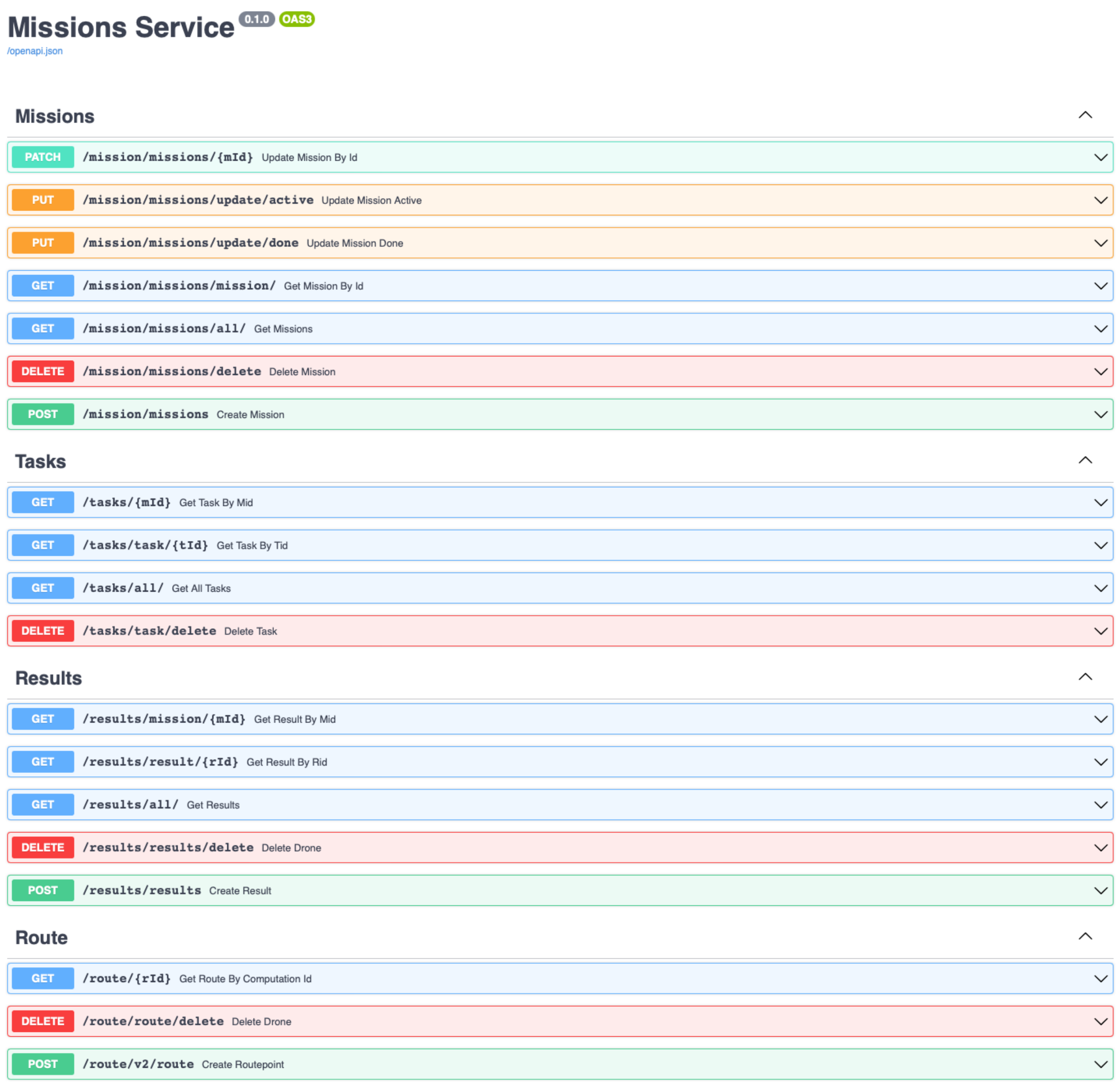
```
        # app
        fastapi
        uvicorn
        numpy
        pydantic

        requests
        python-dotenv # read .evn

        # dev
        pytest
        pytest-asyncio
        pytest-cov
        httpx
        asgi-lifespan
```

With Database

```
    # app
    fastapi
    uvicorn
    pydantic
    starlette-prometheus

    #db
    databases[postgresql]
    SQLAlchemy
    GeoAlchemy2
    psycopg2-binary
    pydantic
    shapely

    requests
    python-dotenv # read .evn

    # dev
    pytest
    pytest-asyncio
    pytest-cov
    httpx
    asgi-lifespan}
```

9.12 Endpoints Listed (Docs)

Mission


Missions Service 0.1.0 OAS3
/openapi.json

Missions

PATCH /mission/missions/{mId} Update Mission By Id
PUT /mission/missions/update/active Update Mission Active
PUT /mission/missions/update/done Update Mission Done
GET /mission/missions/mission/ Get Mission By Id
GET /mission/missions/all/ Get Missions
DELETE /mission/missions/delete Delete Mission
POST /mission/missions Create Mission

Tasks

GET /tasks/{mId} Get Task By Mid
GET /tasks/task/{tId} Get Task By Tid
GET /tasks/all/ Get All Tasks
DELETE /tasks/task/delete Delete Task

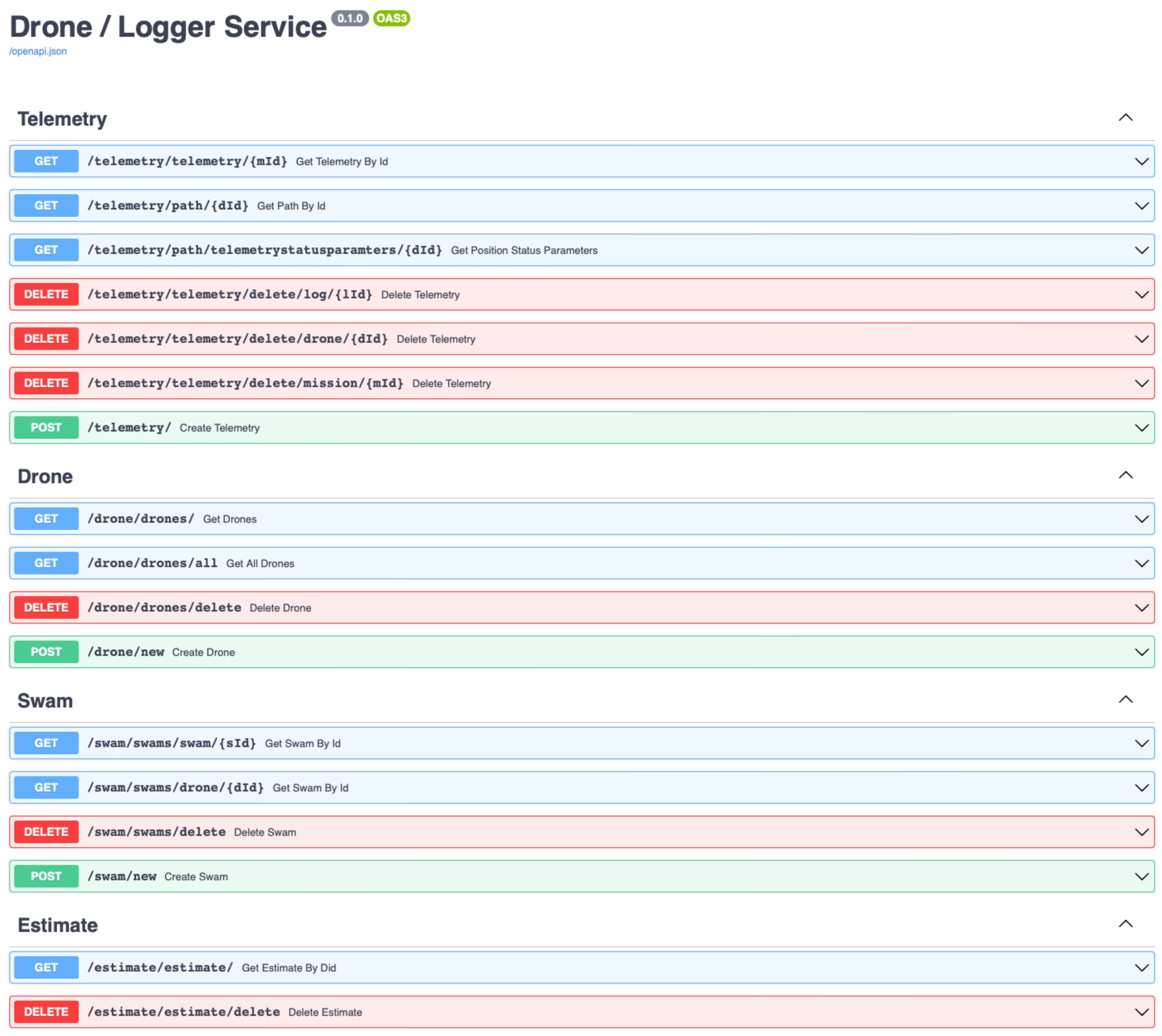
Results

GET /results/mission/{mId} Get Result By Mid
GET /results/result/{rId} Get Result By Rid
GET /results/all/ Get Results
DELETE /results/results/delete Delete Drone
POST /results/results Create Result

Route

GET /route/{rId} Get Route By Computation Id
DELETE /route/route/delete Delete Drone
POST /route/v2/route Create Routepoint

Drone/Logger



Drone / Logger Service 0.1.0 OAS3

/openapi.json

Telemetry

```
GET     /telemetry/telemetry/{mId}  Get Telemetry By Id
GET     /telemetry/path/{dId}  Get Path By Id
GET     /telemetry/path/telemetrystatusparamters/{dId}  Get Position Status Parameters
DELETE  /telemetry/telemetry/delete/log/{lId}  Delete Telemetry
DELETE  /telemetry/telemetry/delete/drone/{dId}  Delete Telemetry
DELETE  /telemetry/telemetry/delete/mission/{mId}  Delete Telemetry
POST    /telemetry/  Create Telemetry
```

Drone

```
GET     /drone/drones/  Get Drones
GET     /drone/drones/all  Get All Drones
DELETE  /drone/drones/delete  Delete Drone
POST    /drone/new  Create Drone
```

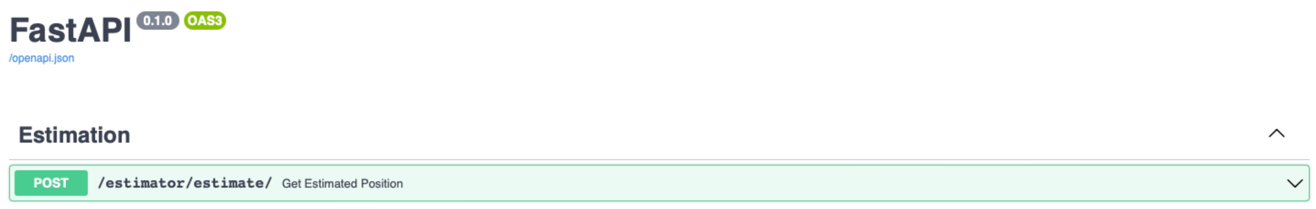
Swam

```
GET     /swam/swams/swam/{sId}  Get Swam By Id
GET     /swam/swams/drone/{dId}  Get Swam By Id
DELETE  /swam/swams/delete  Delete Swam
POST    /swam/new  Create Swam
```

Estimate

```
GET     /estimate/estimate/  Get Estimate By Did
DELETE  /estimate/estimate/delete  Delete Estimate
```

Estimator


FastAPI 0.1.0 OAS3

/openapi.json

Estimation

POST /estimator/estimate/ Get Estimated Position

Bibliography

[1] D. Alex. Sql, nosql, and scale: How dynamodb scales where relational databases don't. https://www.alexdebrie.com/posts/dynamodb-no-bad-queries/, 2020. [Online; accessed 06-12-2021].

[2] G. C. Benjamin, Good (Solutions Architect. To run or not to run a database on Kubernetes: What to consider. https://cloud.google.com/blog/products/databases/to-run-or-not-to-run-a-database-on-kubernetes-what-to-consider?fbclid=IwAR1atcuguYfmocAdlg66_37HsTxGkyFtoTmEypFvUHZo5BT4hA9IdhFPe80, 2019. [Online; accessed 06-12-2021].

[3] S. Brad. Async io in python: A complete walkthrough. https://realpython.com/async-io-python/#the-asyncio-package-and-asyncawait, 2019. [Online; accessed 19-12-2021].

[4] V. Chris. Calculate distance, bearing and more between latitude/longitude points. http://www.movable-type.co.uk/scripts/latlong.html, 2020. [Online; accessed 28-12-2021].

[5] M. David. [question] using fast-api non-async libraries #260. https://github.com/tiangolo/fastapi/issues/260, 2019. [Online; accessed 19-12-2021].

[6] Docker Docs. Dockerfile reference - workdir. https://docs.docker.com/engine/reference/builder/#workdir, 2021. [Online; accessed 19-12-2021].

[7] Docker Docs. Use docker compose. https://docs.docker.com/get-started/08_using_compose/, 2021. [Online; accessed 19-12-2021].

[8] Docker Team. Compose file reference. https://docs.docker.com/compose/compose-file/compose-file-v3/#volumes, 2021. [Online; accessed 19-12-2021].

[9] Docker Team. Dockerize postgresql. https://docs.docker.com/samples/postgresql_service/, 2021. [Online; accessed 19-12-2021].

[10] Docker Team. Use containers to build, share and run your applications. https://www.docker.com/resources/what-container, 2021. [Online; accessed 19-12-2021].

[11] encode/Databases. Databases. https://www.encode.io/databases/, 2021. [Online; accessed 19-12-2021].

[12] encode/starlette. Starlette. https://www.starlette.io, 2021. [Online; accessed 19-12-2021].

[13] encode/uvicorn. Uvicorn. https://www.uvicorn.org, 2021. [Online; accessed 19-12-2021].

[14] Energinet. Energinet. http://geodanmark.nu/Spec6/HTML5/DK/StartHer.htm#GEDS6-DK/3.4.14%20Telemast.htm%3FTocPath%3D3.4%2520TEKNIK%7C_____14, 2021. [Online; accessed 02-12-2021].

[15] Flask. Flask - web developement one drop at a time. https://flask.palletsprojects.com/en/2.0.x/, 2021. [Online; accessed 02-12-2021].

[16] Geoalchemy2 Team. Geoalchemy2. https://geoalchemy-2.readthedocs.io/en/latest/, 2021. [Online; accessed 23-12-2021].

[17] Geoalchemy2 Team. Spatial functions. https://geoalchemy-2.readthedocs.io/en/latest/spatial_functions.html, 2021. [Online; accessed 23-12-2021].

[18] GeoData. GeoData. www.geodata-info.dk, 2021. [Online; accessed 02-12-2021].

[19] M. Golizheh and S.-K. Peter. *Optimal Path Planning for Drone Inspections of Linear Infrastructures.* University of Southern Denmark, Odense Denmark, 2020.

[20] Google. Vehicle Routing Problem. `https://developers.google.com/optimization/routing/vrp`, 2021. [Online; accessed 02-12-2021].

[21] Griffith University, Australia. Database design. `http://www.ict.griffith.edu.au/normalization_tools/normalization/assets/Functional%20Dependencies%20and%20Normalization.pdfv`, -. [Online; accessed 30-12-2021].

[22] G.-M. Hector, U. Jeffrey, D., and W. Jennifer. *Database Systems. The Complete Book.* University of Southern Denmark, Odense Denmark, 2021.

[23] Holger, Krekel and pytest-dev team. pytest: helps you write better programs. `https://docs.pytest.org/en/6.2.x/`, 2020. [Online; accessed 02-12-2021].

[24] Instagram Engineering. Sharding & ids at instagram. `https://instagram-engineering.com/sharding-ids-at-instagram-1cf5a71e5a5c`, 2012. [Online; accessed 30-12-2021].

[25] A. Kazantcev. How to send 204 response? #717. `https://github.com/tiangolo/fastapi/issues/717`, 2019. [Online; accessed 23-12-2021].

[26] S. Kevin. How do websockets work? `https://sookocheff.com/post/networking/how-do-websockets-work/`, 2019. [Online; accessed 30-12-2021].

[27] P. Konstantin and Y. Olga. Minimum viable product in software development: Getting it right. `https://codetiburon.com/minimum-viable-product-in-software-development-getting-it-right/`, 2019. [Online; accessed 30-12-2021].

[28] M. Lea. *From Monolith to Microservice: Software Architecture for Autonomous UAV Infrastructure Inspection.* University of Southern Denmark, Odense Denmark, 2021.

[29] M. Lea. Specification of the drone inspection as a service platform (m6). Unpublished Specification, 2021.

[30] Lyn, Muldrow. What is dry development? `https://www.digitalocean.com/community/tutorials/what-is-dry-development`, 2020. [Online; accessed 30-12-2021].

[31] O. S. Map. Vehicle Routing Problem. `VehicleRoutingProblem`, 2021. [Online; accessed 02-12-2021].

[32] F. Martin. Flask - web developement one drop at a time. `https://martinfowler.com/bliki/MonolithFirst.html`, 2015. [Online; accessed 02-12-2021].

[33] F. Martin. Testing Strategies in a Microservice Erchitcture. `https://martinfowler.com/articles/microservice-testing/#testing-integration-diagraml`, 2015. [Online; accessed 02-12-2021].

[34] F. H. Mathis. *A Generalized Birthday Problem.* Society for Industrial and Applied Mathematics, Philadelphia, 1990.

[35] Microsoft. Common web application architectures. `https://www.alexdebrie.com/posts/dynamodb-no-bad-queries/`, 2021. [Online; accessed 16-12-2021].

[36] Microsoft. Common web application architectures. `https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures`, 2021. [Online; accessed 30-12-2021].

[37] B. S. Mike. Asynchronous python and databases. `https://techspot.zzzeek.org/2015/02/15/asynchronous-python-and-databases/`, 2020. [Online; accessed 19-12-2021].

[38] MongoDB. Geospatial Queries. `https://docs.mongodb.com/manual/geospatial-queries/`, 2021. [Online; accessed 02-12-2021].

[39] MongoDB. MongoDB Atlas. `https://www.mongodb.com/atlas/database`, 2021. [Online; accessed 02-12-2021].

[40] Mozilla. An overview of HTTP. https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview, 2021. [Online; accessed 02-12-2021].

[41] NetworkX developers. Networkx. https://networkx.org, 2021. [Online; accessed 06-12-2021].

[42] Numpy Team. numpy.errstate. https://numpy.org/doc/stable/reference/generated/numpy.errstate.html, 2019. [Online; accessed 28-12-2021].

[43] Numpy Team. numpy.nan_to_num. https://numpy.org/doc/stable/reference/generated/numpy.nan_to_num.html#numpy-nan-to-num, 2019. [Online; accessed 28-12-2021].

[44] OpenStreetMap. EnerOpenStreetMapginet. https://www.openstreetmap.org, 2021. [Online; accessed 02-12-2021].

[45] T. Paolo and V. Daniele. *The vehicle routing problem.* The vehicle routing problem, Philadelphia, 2002.

[46] PostGIS. About postgis. https://postgis.net, 2020. [Online; accessed 19-12-2021].

[47] PostGIS. Enabling postgis. https://postgis.net/install/, 2020. [Online; accessed 19-12-2021].

[48] PostGIS. Introduction to postgis. https://postgis.net/workshops/postgis-intro/introduction.html, 2020. [Online; accessed 19-12-2021].

[49] PostgreSQL. Why use postgresql? https://www.postgresql.org/about/, 2020. [Online; accessed 19-12-2021].

[50] PostgreSQL. Postgresql database management system. https://github.com/postgres/postgres, 2021. [Online; accessed 19-12-2021].

[51] Python Docs. 3.3.9. with statement context managers. https://docs.python.org/3/reference/datamodel.html#with-statement-context-managers, 2018. [Online; accessed 18-12-2021].

[52] Python Docs. 8.5. user-defined exceptions. https://docs.python.org/3.3/tutorial/errors.html#tut-userexceptions, 2018. [Online; accessed 18-12-2021].

[53] Python Docs. Rich comparison methods. https://docs.python.org/3/reference/datamodel.html#object.__lt__, 2018. [Online; accessed 28-12-2021].

[54] Raymond, J., Rubey. Spaghetti code. https://www.gnu.org/fun/jokes/pasta.code.html, 1992. [Online; accessed 30-12-2021].

[55] C. Samuel. Custom root types. https://pydantic-docs.helpmanual.io/usage/models/#custom-root-types, 2020. [Online; accessed 18-12-2021].

[56] C. Samuel. Pydantic. https://pydantic-docs.helpmanual.io, 2021. [Online; accessed 18-12-2021].

[57] C. Samuel. Settings management. https://pydantic-docs.helpmanual.io/usage/settings/, 2021. [Online; accessed 18-12-2021].

[58] V. Santiago. Caching in python using the lru cache strategy. https://realpython.com/lru-cache-python/, 2021. [Online; accessed 18-12-2021].

[59] Scrum.org. What is scrum? https://www.scrum.org/resources/what-is-scrum, 2012. [Online; accessed 30-12-2021].

[60] R. t. Sebastián. Full stack fastapi and postgresql - base project generator. https://github.com/tiangolo/full-stack-fastapi-postgresql, 2019. [Online; accessed 18-12-2021].

[61] R. t. Sebastián. Background tasks. https://fastapi.tiangolo.com/tutorial/background-tasks/, 2021. [Online; accessed 19-12-2021].

[62] R. t. Sebastián. Concurrency and async / await. https://fastapi.tiangolo.com/tutorial/testing/#testing, 2021. [Online; accessed 02-12-2021].

[63] R. t. Sebastián. Concurrency and async /await. https://fastapi.tiangolo.com/async/, 2021. [Online; accessed 19-12-2021].

[64] R. t. Sebastián. Dependencies - first steps. https://fastapi.tiangolo.com/tutorial/dependencies/#dependencies-first-steps, 2021. [Online; accessed 19-12-2021].

[65] R. t. Sebastián. Dependencies with yield. https://fastapi.tiangolo.com/tutorial/dependencies/dependencies-with-yield/, 2021. [Online; accessed 18-12-2021].

[66] R. t. Sebastián. Events: startup - shutdown. https://fastapi.tiangolo.com/advanced/events/, 2021. [Online; accessed 18-12-2021].

[67] R. t. Sebastián. FastAPI. https://fastapi.tiangolo.com, 2021. [Online; accessed 02-12-2021].

[68] R. t. Sebastián. FastAPI in Containers - Docker. https://fastapi.tiangolo.com/deployment/docker/, 2021. [Online; accessed 02-12-2021].

[69] R. t. Sebastián. Override the httpexception error handler. https://fastapi.tiangolo.com/tutorial/handling-errors/#override-the-httpexception-error-handler, 2021. [Online; accessed 19-12-2021].

[70] R. t. Sebastián. Server Workers - Gunicorn with Uvicorn. https://fastapi.tiangolo.com, 2021. [Online; accessed 02-12-2021].

[71] R. t. Sebastián. Server Workers - Gunicorn with Uvicorn. https://fastapi.tiangolo.com/deployment/server-workers/#uvicorn-with-workers, 2021. [Online; accessed 02-12-2021].

[72] R. t. Sebastián. Server Workers - Gunicorn with Uvicorn. https://fastapi.tiangolo.com/deployment/docker, 2021. [Online; accessed 02-12-2021].

[73] R. t. Sebastián. Websockets. https://fastapi.tiangolo.com/advanced/websockets/, 2021. [Online; accessed 19-12-2021].

[74] t. Sebastián, Ramírez. Concurrency and async / await. https://fastapi.tiangolo.com/async/, 2021. [Online; accessed 02-12-2021].

[75] Shapely Team. Shapely. https://shapely.readthedocs.io/en/stable/, 2021. [Online; accessed 23-12-2021].

[76] B. Sneha, G. Ajinkya, and U. Shiva. SQL vs. NoSQL vs. NewSQL- A Comparative Study. https://caeaccess.com/archives/volume6/number1/binani-2016-cae-652418.pdf, 2016. [Online; accessed 02-12-2021].

[77] SQLAlchemy. Column element modifier constructors. https://docs.sqlalchemy.org/en/14/core/sqlelement.html#sqlalchemy.sql.expression.desc, 2021. [Online; accessed 23-12-2021].

[78] SQLAlchemy. Object relational tutorial (1.x api). https://docs.sqlalchemy.org/en/14/orm/tutorial.html, 2021. [Online; accessed 19-12-2021].

[79] SQLAlchemy. Session basics api. https://docs.sqlalchemy.org/en/14/orm/session_basics.html, 2021. [Online; accessed 20-12-2021].

[80] SQLAlchemy. Sqlalchemy. https://www.sqlalchemy.org, 2021. [Online; accessed 23-12-2021].

[81] SQLAlchemy. Sqlalchemy - changing compilation of types. https://docs.sqlalchemy.org/en/14/core/compiler.html#changing-compilation-of-types, 2021. [Online; accessed 18-12-2021].

[82] SQLAlchemys. Basic relationship patterns. https://docs.sqlalchemy.org/en/14/orm/basic_relationships.html#one-to-many, 2021. [Online; accessed 18-12-2021].

[83] SQLAlchemys. Configuring delete/delete-orphan cascade. https://docs.sqlalchemy.org/en/14/orm/tutorial.html#tutorial-delete-cascade, 2021. [Online; accessed 18-12-2021].

[84] SQLAlchemys. Query api. https://docs.sqlalchemy.org/en/14/orm/query.html, 2021. [Online; accessed 20-12-2021].

[85] SQLAlchemys. Relationship loading techniques. https://docs.sqlalchemy.org/en/14/orm/loading_relationships.html#, 2021. [Online; accessed 20-12-2021].

[86] SQLAlchemys. Sqlalchemy. https://docs.sqlalchemy.org/en/14/core/engines.html, 2021. [Online; accessed 18-12-2021].

[87] Stefan, Behnel. About cython. https://cython.org, 2021. [Online; accessed 19-12-2021].

[88] TechEmpower. Web Framework Benchmarks - Round 20. https://www.techempower.com/benchmarks/, 2021. [Online; accessed 02-12-2021].

[89] S. R. (@tiangolo). JSON Compatible Encoder. https://fastapi.tiangolo.com/tutorial/encoder/, 2021. [Online; accessed 22-12-2021].

[90] The Psycopg Team. Psycopg. https://www.psycopg.org, 2020. [Online; accessed 19-12-2021].

[91] Tom, Harrison. Uuid or guid as primary keys? be careful! https://tomharrisonjr.com/uuid-or-guid-as-primary-keys-be-careful-7b2aa3dcb439, 2017. [Online; accessed 30-12-2021].

[92] S. Yury. uvloop: Blazing fast python networking. https://cython.org, 2016. [Online; accessed 19-12-2021].