Distributed Systems Engineering Submission Document SUPD (Status Update)

Team member 1	
Name:	Nenad Cikojevic
Student ID:	01549084
E-mail address:	a01549084@unet.univie.ac.at

Team member 2	
Name:	Nemanja Srdanovic
Student ID:	01576891
E-mail address:	a01576891@unet.univie.ac.at

Team member 3	
Name:	Adem Mehremic
Student ID:	01650669
E-mail address:	a01650669@unet.univie.ac.at

Team member 4	
Name:	Veljko Radunovic
Student ID:	01528243
E-mail address:	a01528243@unet.univie.ac.at

API Specification: Detailed definition of the network/microservice API

MSCF API Specification (Peer 2 Peer Communication Framework):

Technology:

Programming language: Java11, HTML5 + Javascript

IDE: Eclipse 2019-12

Build tool: Maven

Database: SQLite

Communication protocol: UDP, REST

Introduction: MSCF is intended to be a framework that keeps entire communication and message- exchange logic “under the hood”. It is wrapped inside of a Java library which is imported by every MS. Its functionalities are separated in packages which will be further described.

1. IP scanning: Due to required functionality that each MS should work on every machine and the IP address should not be constant. MSCF provides Network interfaces scanning when MS starts by comparing the previously provided list of all IP addresses of MSs. When the VPN Address and of the machine is found it is kept as an instance of Node class. The logic behind IP Scanning is kept in the IPAdressHandler class.

2.Simulation of dynamic network conditions: To simulate real world conditions where networks are not stable, every MS that implements MSCF will be automatically reconnected with two randomly chosen MSs in every 10 seconds. In order to implement this functionality, we have Created class ConnectionHandler which extends TimerTask.

3.Routing and forwarding: Each MSs has 2 Connection requesters (UDPClient class)- each requests connection with randomly chosen MS and one Listener (UDPServer class)- waits for other MSs to connect. At the very beginning, they (requesters) send a Greetings message to connected MS and receive as a response it’s IP address, port and MSid. For the sending - MS checks if it is directly connected to the message “destination”. If so, it sends a message directly to the recipient. Otherwise, the message will be sent to both connected MSs which in that case forward these messages to their connected Nodes (if one of the Nodes is not the sender of the message). Cases in which destination MS is down, or a Network overload has happened is solved via TTL (time to live) inside of a “package” (Message object) which will be dropped when the time to live expires.

4. Persistent Message storage: To enable MS to save Messages and resend them again, in case a microservice is down or a network overloaded, a database on each microservice has been implemented. Every Listener (UDPServer) saves the message inside the database at message arrival. The messages are then extracted by the MessageHendler, which will, based on their destination, decide if the message will be forwarded to the UDPClient or processed on the current MS. When one of those two options is successfully executed the message will be put back in the database and marked depending on the option, so that it can be used for statistical purposes.

5.Distinguishing between message usage: In order to put different class types inside a single message type, we decided to hold the data as an Object type and wrap it in a Message object. Data types that "travel" between MSs are going to be explained in the next paragraphs.

6. Exchanged data/messages:

The exchange of data between the microservices is defined in the mscf and done over UDP sockets. Therefore, all generated data is put inside a message object, which is then converted to a Datagram over the Marshaller class and sent over the UDPClient to an available microservice. All the exchanged messages are more detailed explained in the MSx API Specifications below.

7. Endpoints:

Endpoints are constructed as identifiers for every microservice and are hardcoded inside the different microservice class. They are used to invoke a specific function inside the MS which has to process the message data in a specific way. Each message has the name of the endpoint it is intended for, and with it also the microservice which is the destination for this message. The endpoint identifiers are different for every microservice, and therefore the microservice has only to prove the endpoint to know if the message is intended for it to process.

8. Resilience and Error handling:

We have covered following cases that will occur and fit under the topic "Resilience and Error handling":

- *New Message(UUID doesn't exist in DB)* UDPServer listens for multiple requests in parallel and answers "OK" when the message is received. Decides what to do with a message (Forward/ Process) and places it into the DB (with status- "Process", "Forward"). Messages to forward are put into the Queue (List) which are sent by UDPClient. Messages to process are unpacked and handed to MS for further processing.
- *Received message exists in DB (existing UUID)*
    a. if the MS is the destination of this message ("Process" status in DB)- this message is considered already taken into processing so it is duplicate- it can be forgotten
    b. if the MS is not final destination and
        1. the time it is received differs from time sent more than 5 seconds change status to LOST

        2. and the time it is received differs from time sent less than 5 send again

- *After sending a message UDPClient should wait for (e.g 300 milliseconds)* an “OK” answer from MS recipient.
    1. If the answer is received, message with UUID (correlation ID, unique id for every message) is taken from DB and the time stamp and “sent” is set as an attribute.
    2. Otherwise, repeats one more time (sends and waits). If the second time was also unsuccessful it is taken from DB and the time stamp and “not sent” is set as an attribute.

Two cases can cause an Error:

- expected data type is not received
- received datatype is corrupted

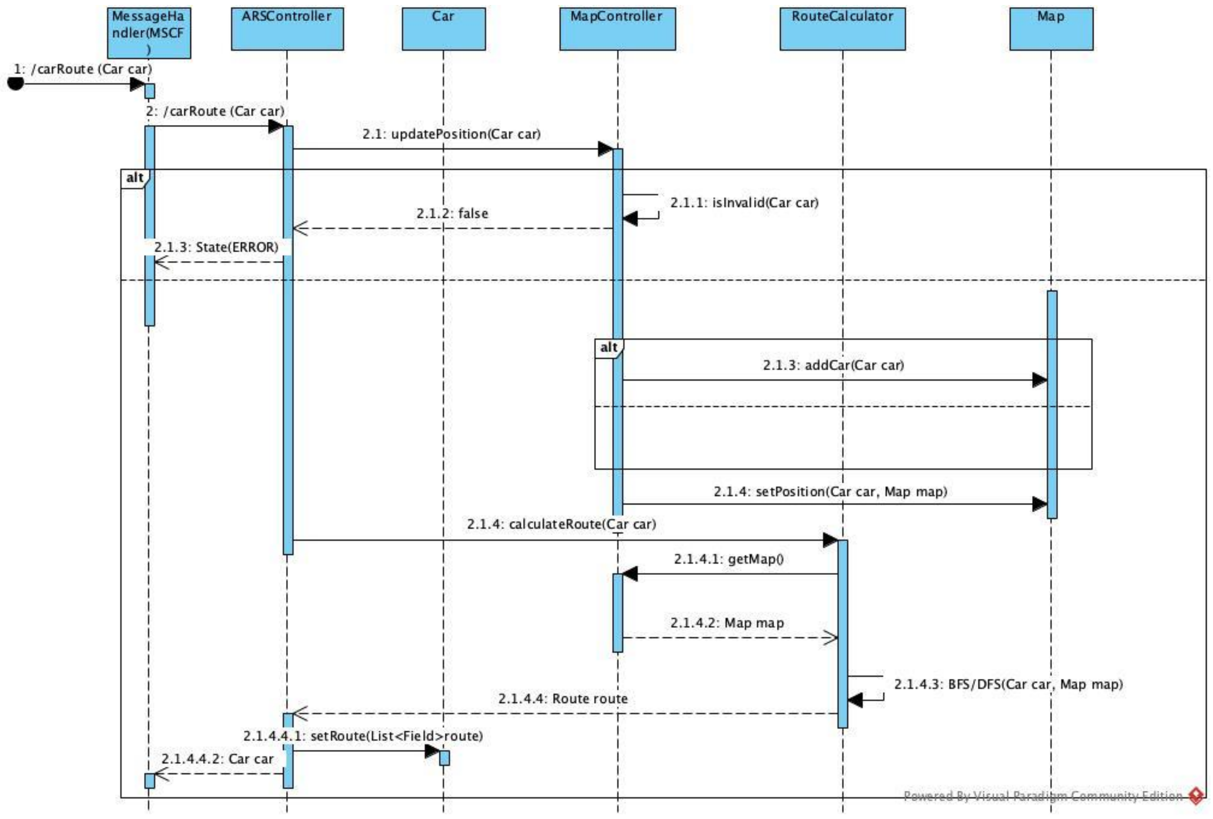
In both cases appropriate exceptionName and exceptionMessage should be wrapped inside Response message and returned to sender.

MS1 API Specification (Automatic Routing Service):

The MS1 is used to generate route for self- driving cars. Generating of the route is calculated by the combination of the Breadth/Depth First Search algorithm which will be explained in detail later. It also holds the data about the City Map, Streets, Cars which will be provided for (and updated by providing the data from) other micro-services. All requests and responses are done over Datagram Sockets which represent the mechanism for network communication via UDP. All the logic for the package exchanging is define in the MSCF that is imported by the micro- services. The following sequence diagrams and use cases represent the package exchange between the Automatic Routing Service and the others micro-services inside the network.

<u>Use cases and associated sequence diagrams</u>

1. Sequence diagram “Get route for a given Car”:



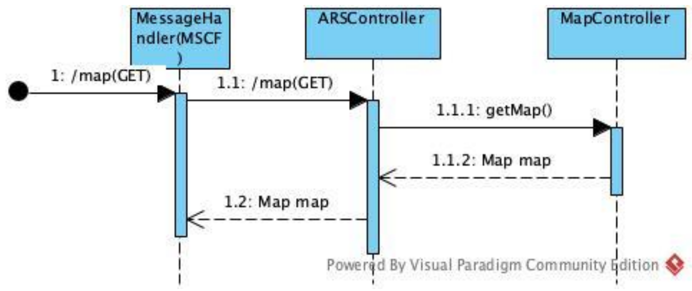
Title	Get route for a given Car
Description and Use Case	Calculate the route for the given car object received at udp server. When received the MS1 checks if the object already exists inside cache. Use case 1 - Received new car object • Object will be inserted in cache (List<Car>) • The new car object will be put inside a List<Car> on the Field which has the same coordinates as the car position. • Then the RouteCalculator class will be triggered, so that a route from the car's position to the car´s destination will be calculated. • The calculated route will be put inside the car object, and the object sent to the destination (MS2) it came from, as a Message object. Use case 2 - Received an existing car object • For the existing car object the position and destination will be set accordingly with the received car object. • The car position will be adapted on the map accordingly with the received object. • All other steps are exactly as in the use case 1.
Transport Protocol Details	UDP Client, endpoint: /carRoute UDP Server, endpoint: /carRoute

Sent Body Example
 	
(Client)
	{
	    "MessageId": "21",
	    "SourceIP": "10.101.101.13",
	    "DestinationIP": "10.101.101.5",
	    "DestinatedMS": "MS1/carRoute",
	    "Status": "null",
	    "VisitedMS": "|MS2|",
	    "TimeStamp": "2016-11-16 06:43:19.77",
	    "Object": "Car",
	}

Sent Body Description	▌MessageId: Message identifier which is unique and automatically assigned. The first number is always the number of the microservice. ▌SourceIP: The IP address of the microservice which generated the message. ▌DestinationIP: The IP address of the next hop microservice, or in best case the destination microservice. ▌DestinatedMS: Endpoint identifier for the destination microservice.

	Status: Status of the executed process on the message in the current microservice. **VisitedMS**:List of visited microservices till arrival at the destination. **TimeStamp**: Date and time of the arrival at the microservice. **Object**: The data which has to be processed at the destination.
Response Body Example **(Server)**	`{ "MessageId": "11", "SourceIP": "10.101.101.5", "DestinationIP": "10.101.101.13", "DestinatedMS": "MS2/carRoute", "Status": "null", "VisitedMS": "|MS1|", "TimeStamp": "2016-11-16 06:43:25.77", "Object": "Car", }`
Response Body Description	**MessageId**: Message identifier which is unique and automatically assigned. The first number is always the number of the microservice. **SourceIP**: The IP address of the microservice which generated the message. **DestinationIP**: The IP address of the next hop microservice, or in best case the destination microservice. **DestinatedMS**: Endpoint identifier for the destination microservice. **Status**: Status of the executed process on the message in the current microservice. **VisitedMS**:List of visited microservices till arrival at the destination. **TimeStamp**: Date and time of the arrival at the microservice. **Object**: The data which has to be processed at the destination.
Error Cases	**Position/Destination field invalid**: The position or destination field type is not drivable or invalid (grass), so that the car route can not be calculated. Also it could be that the maximum amount of cars on a field is already reached. **Invalid data type**: Expected data type is Car in every other case exception is thrown.
Notes & Remarks	The car object is further described in MSCF Description/Messages.

2. Sequence diagram “Get Map”:



Title	Map requested (list of all streets)
Description and Use Case	The microservice MS4 sends a request to get a map object with a list of all available streets and their fields, so that the requester can work with those objects. <u>Use case - Receiving a map request from MS4:</u> • The map object will be packed into a message object and sent to the requester (MS4)
Transport Protocol Details	UDP Client, endpoint: /map UDP Server, endpoint: /map

Sent Body Example	{
	"MessageId": "11",
	"SourceIP": "10.101.101.17",
	"DestinationIP": "10.101.101.5",
	"DestinatedMS": "MS1/map",
	"Status": "null",
	"VisitedMS": "|MS4|MS2|",
	"TimeStamp": "2016-11-16 06:45:19.77",
	"Object": "null",
	}

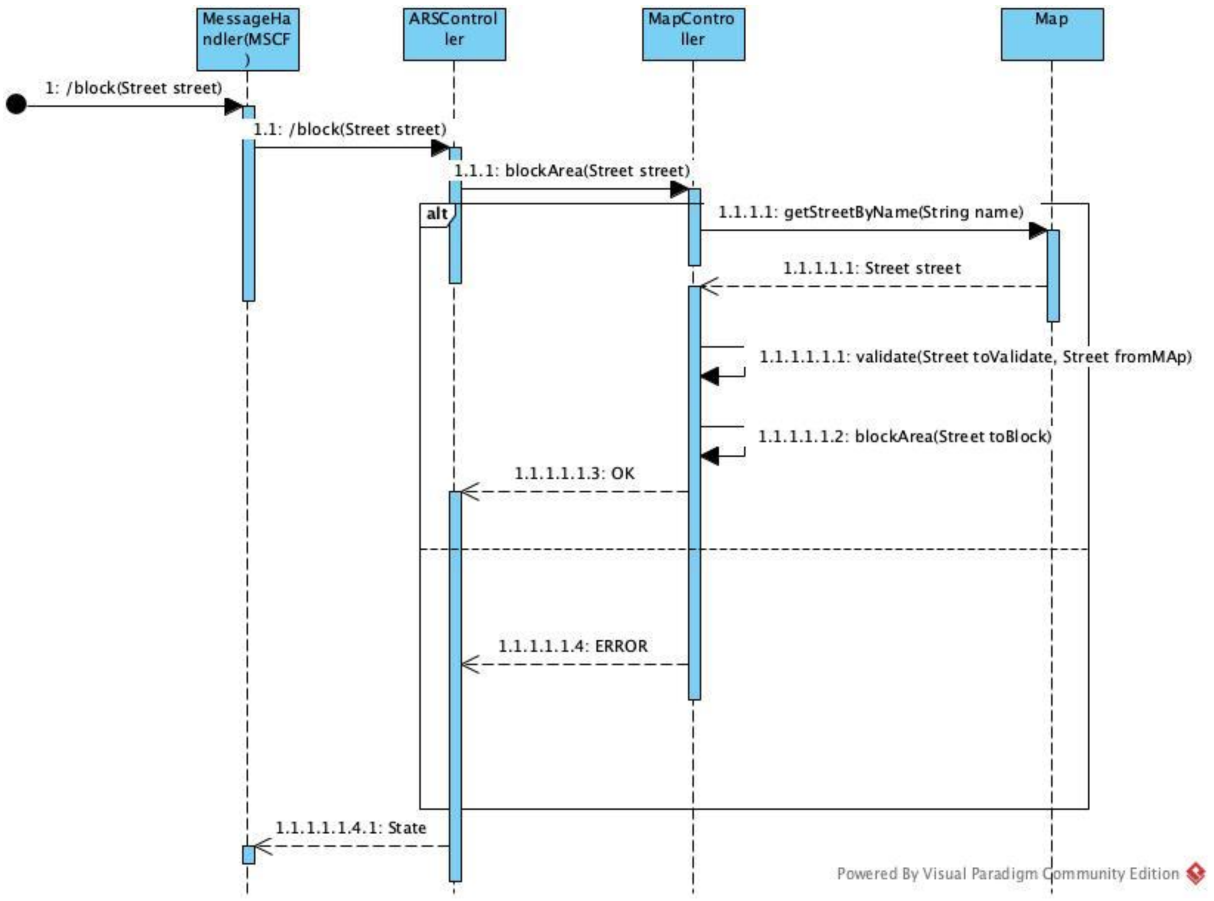
Sent Body Description	l MessageId: Message identifier which is unique and automatically assigned. The first number is always the number of the microservice. l SourceIP: The IP address of the microservice which generated the message. l DestinationIP: The IP address of the next hop microservice, or in best case the destination microservice. l DestinatedMS: Endpoint identifier for the destination microservice.

	Status: Status of the executed process on the message in the current microservice. VisitedMS:List of visited microservices till arrival at the destination. TimeStamp: Date and time of the arrival at the microservice. Object: Is empty cause no object needed for the request.

Response Body Example	`{`
	`"MessageId": "12",`
	`"SourceIP": "10.101.101.5",`
	`"DestinationIP": "10.101.101.17",`
	`"DestinatedMS": "MS4/map",`
	`"Status": "null",`
	`"VisitedMS": "|MS1|",`
	`"TimeStamp": "2016-11-16 06:45:25.77",`
	`"Object": "Map",`
	`}`

Response Body Description	MessageId: Message identifier which is unique and automatically assigned. The first number is always the number of the microservice. SourceIP: The IP address of the microservice which generated the message. DestinationIP: The IP address of the next hop microservice, or in best case the destination microservice. DestinatedMS: Endpoint identifier for the destination microservice. Status: Status of the executed process on the message in the current microservice. VisitedMS:List of visited microservices till arrival at the destination. TimeStamp: Date and time of the arrival at the microservice. Object: The data which has to be processed at the destination.
Error Cases	Invalid endpoint: If an endpoint placed in DestinatedMS field does not match the predefined microservice endpoint, the message will be forwarded as long the time-to-live has not expired.
Notes & Remarks	The map object is further described in MSCF Description/Messages.

3. Sequence diagram “Block requested area”:


MessageHandler(MSCF)
ARSController
MapController
Map
1: /block(Street street)
1.1: /block(Street street)
1.1.1: blockArea(Street street)
alt
1.1.1.1: getStreetByName(String name)
1.1.1.1.1: Street street
1.1.1.1.1.1: validate(Street toValidate, Street fromMAp)
1.1.1.1.1.2: blockArea(Street toBlock)
1.1.1.1.1.3: OK
1.1.1.1.1.4: ERROR
1.1.1.1.1.4.1: State
Powered By Visual Paradigm Community Edition

Title	Block requested area
Description and Use Case	Receives a request containing an specific area that should be blocked (Fields marked undrivable). The area can be a whole Street or just a part of it. Use case: • Receives an Street object in a message object • A list inside the Street object containing Field objects, is verified to make sure they are of type street. • If okay, the Fields from the Street object will be found on the map and their flag updated to undrivable(false). • After the update is done, an Map fresh map object with the changed Fields will be sent to the requester.
Transport Protocol Details	UDP, endpoint: /block

Sent Body Example	`{`
	`    "MessageId": "42",`
	`    "SourceIP": "10.101.101.17",`
	`    "DestinationIP": "10.101.101.5",`
	`    "DestinatedMS": "MS1/block",`
	`    "Status": "null",`
	`    "VisitedMS": "|MS4|",`
	`    "TimeStamp": "2016-11-16 06:46:19.77",`
	`    "Object": "Street",`
	`}`

Sent Body Description	- MessageId: Message identifier which is unique and automatically assigned. The first number is always the number of the microservice. - SourceIP: The IP address of the microservice which generated the message. - DestinationIP: The IP address of the next hop microservice, or in best case the destination microservice. - DestinatedMS: Endpoint identifier for the destination microservice. - Status: Status of the executed process on the message in the current microservice. - VisitedMS:List of visited microservices till arrival at the destination. - TimeStamp: Date and time of the arrival at the microservice. - Object: The data which has to be processed at the destination.

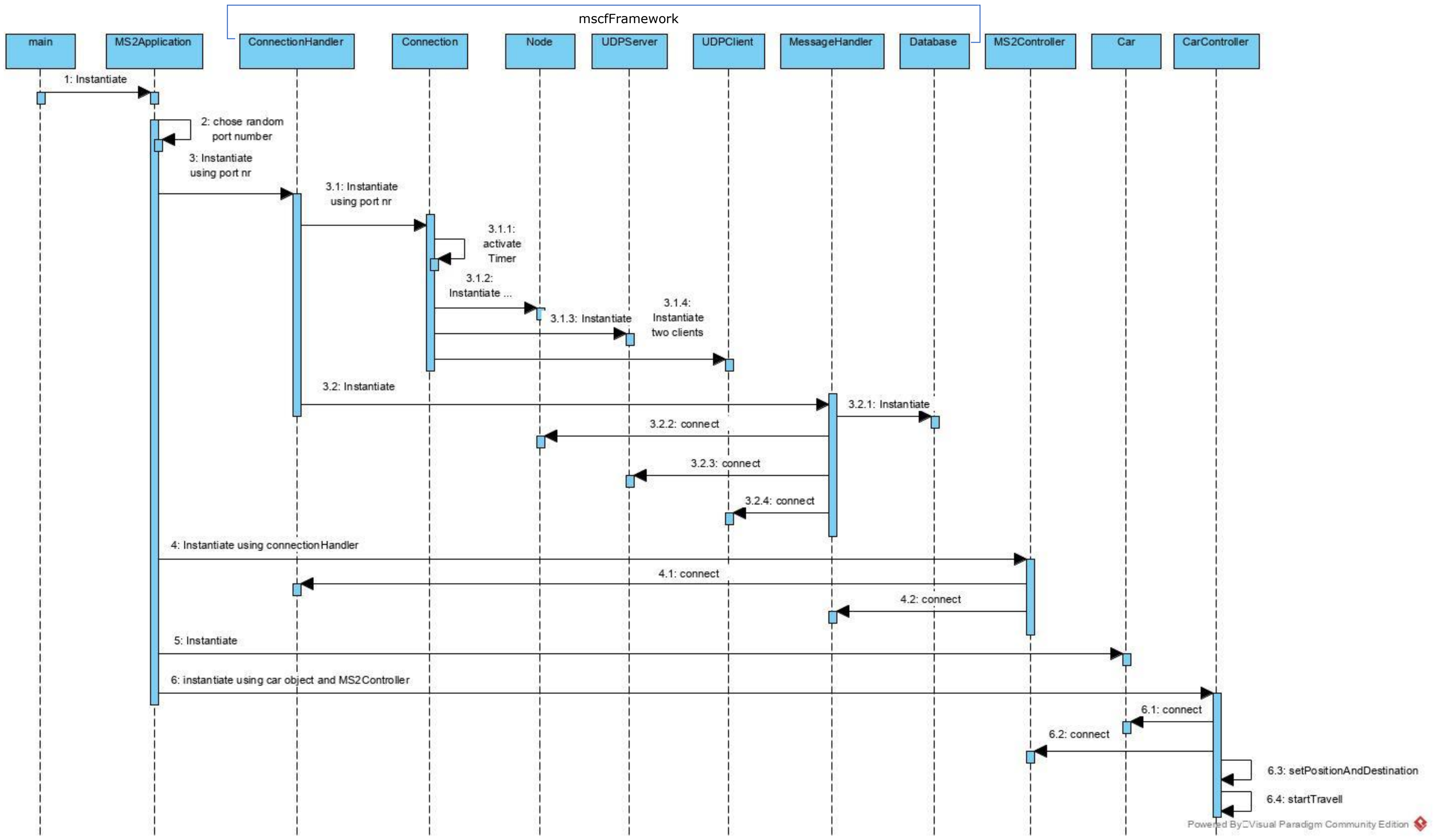
Response Body Example	`{`
	`    "MessageId": "13",`
	`    "SourceIP": "10.101.101.5",`
	`    "DestinationIP": "10.101.101.13",`
	`    "DestinatedMS": "MS4/blockState",`
	`    "Status": "null",`
	`    "VisitedMS": "|MS1|",`
	`    "TimeStamp": "2016-11-16 06:47:25.77",`
	`    "Object": "Map",`
	`}`

Response Body Description	- MessageId: Message identifier which is unique and automatically assigned. The first number is always the number of the microservice. - SourceIP: The IP address of the microservice which generated the message. - DestinationIP: The IP address of the next hop microservice, or in best case the destination microservice. - DestinatedMS: Endpoint identifier for the destination microservice. - Status: Status of the executed process on the message in the current microservice. - VisitedMS:List of visited microservices till arrival at the destination. - TimeStamp: Date and time of the arrival at the microservice. - Object: The data which has to be processed at the destination.
Error Cases	- Received Area(Street and the Fields)is invalid: If a received Object is not an instance of Street, Field, or Coordinate. If the FieldType is not *STREET* - Non of received Fields to (un)block in the list is set to (un)drivable
Notes & Remarks	The street object is further described in MSCF Description/Messages.

MS2 API Specification (Car Data Gateway):

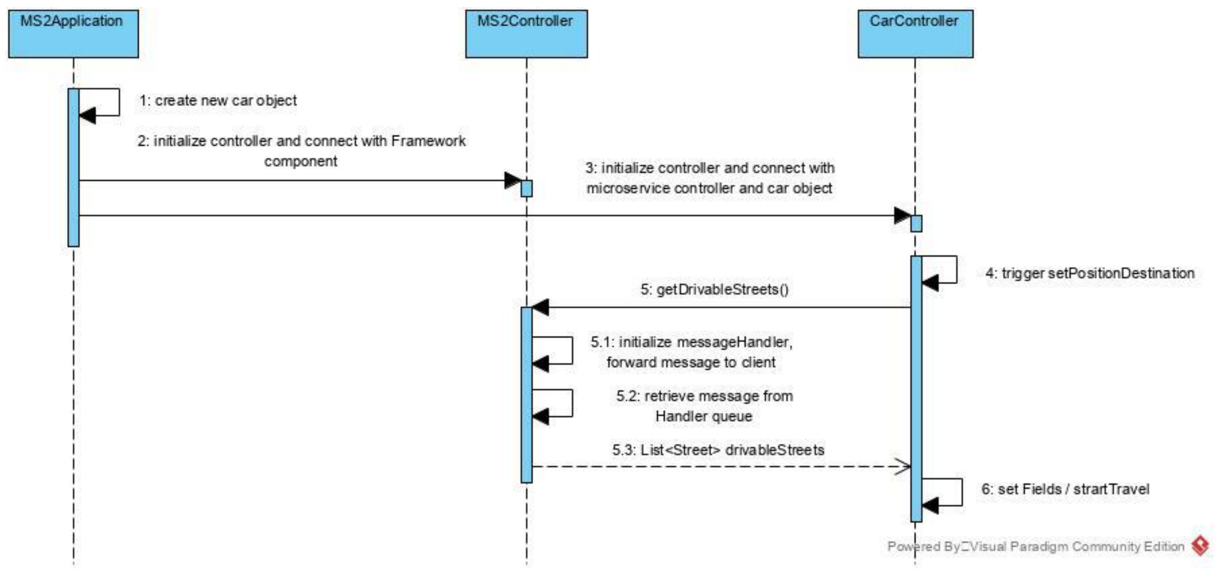
The MS2 microservice represents a minimalistic service which simulates a self-driving car. It simulates the traveling along a route provided by the Automatic Routing Service. For the route calculation it provides the ARS with an car object containing a current position and wanted destination. All requests and responses are done over DatagramSockets which represent the mechanism for network communication via UDP. All the logic for the package exchanging is define in the mscfFramework that is imported by the microservice. The following sequence diagrams and use cases represent the instantiating process of the microservice as the package exchange between the Car Data Gateway and the others microservices inside the network.

Instantiating the microservice:



Use cases and associated sequence diagrams

1. Sequence diagram "Request drivable Streets":


MS2Application
MS2Controller
CarController
1: create new car object
2: initialize controller and connect with Framework component
3: initialize controller and connect with microservice controller and car object
4: trigger setPositionDestination
5: getDrivableStreets()
5.1: initialize messageHandler, forward message to client
5.2: retrieve message from Handler queue
5.3: List<Street> drivableStreets
6: set Fields / strartTravel
Powered By Visual Paradigm Community Edition

1.2 Use case and message description

Title	Request drivable Streets
Description and Use Case	To set the cars position and destination, the microservice requests a List of all drivable Fields. Use case: • The microservice creates a new Car object and initializes the function to set the objects position and destination. For this the microservice needs a List of all drivable fields. • Sends a get request in form of a message where the object is null. • Receives a message with the List<Street> and choses random the position and destination, after which he triggers the function travelRoute.
Transport Protocol Details	UDP, Client endpoint: /drivableFields UDP, Server endpoint: /drivableFields

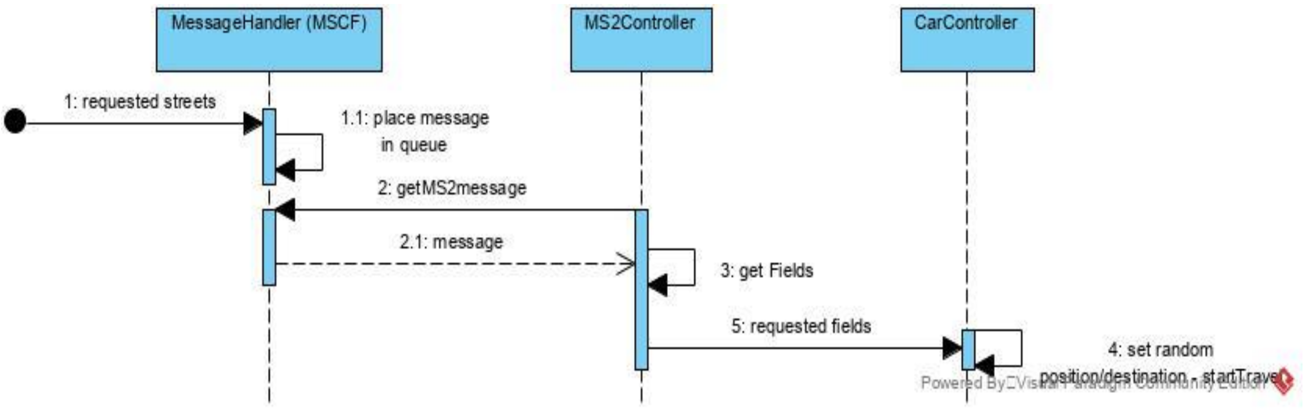
Sent Body Example (Client)	{
	    "MessageId": "22",
	    "SourceIP": "10.101.101.13",
	    "DestinationIP": "10.101.101.5",
	    "DestinatedMS": "MS1/drivableFields",
	    "Status": "null",
	    "VisitedMS": "|MS2|",
	    "TimeStamp": "2016-11-16 06:49:19.77",
	    "Object": "null",
	}

Sent Body Description	❙ MessageId: Message identifier which is unique and automatically assigned. The first number is always the number of the microservice. ❙ SourceIP: The IP address of the microservice which generated the message. ❙ DestinationIP: The IP address of the next hop microservice, or in best case the destination microservice. ❙ DestinatedMS: Endpoint identifier for the destination microservice. ❙ Status: Status of the executed process on the message in the current microservice. ❙ VisitedMS:List of visited microservices till arrival at the destination. ❙ TimeStamp: Date and time of the arrival at the microservice. ❙ Object: Is empty cause no object needed for the request.

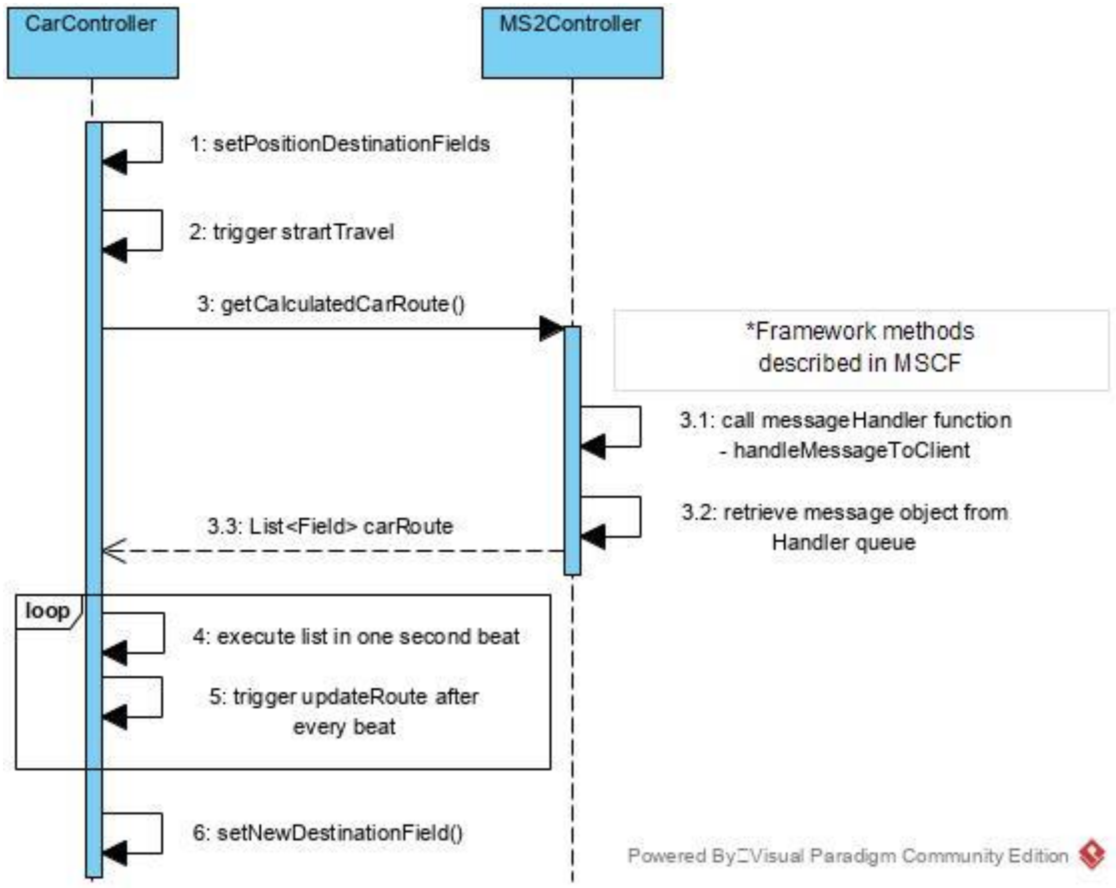
Response Body Example (Server)	{
	    "MessageId": "14",
	    "SourceIP": "10.101.101.5",
	    "DestinationIP": "10.101.101.13",
	    "DestinatedMS": "MS2/blockState",
	    "Status": "null",
	    "VisitedMS": "|MS1|",
	    "TimeStamp": "2016-11-16 06:50:25.77",
	    "Object": "List<Street>",
	}

Response Body Description	▎MessageId: Message identifier which is unique and automatically assigned. The first number is always the number of the microservice. ▎SourceIP: The IP address of the microservice which generated the message. ▎DestinationIP: The IP address of the next hop microservice, or in best case the destination microservice. ▎DestinatedMS: Endpoint identifier for the destination microservice. ▎Status: Status of the executed process on the message in the current microservice. ▎VisitedMS:List of visited microservices till arrival at the destination. ▎TimeStamp: Date and time of the arrival at the microservice. ▎Object: The data which has to be processed at the destination.
Error Cases	▎Invalid data type: Expected data type is List<Field> in every other case exception is thrown.
Notes & Remarks	The travel method is further described in code implementation.

1.3 Sequence diagram - Response message



2. Sequence diagram “Request car route”:


CarController
MS2Controller
1: setPositionDestinationFields
2: trigger strart Travel
3: getCalculatedCarRoute()
*Framework methods described in MSCF
3.1: call messageHandler function - handleMessageToClient
3.2: retrieve message object from Handler queue
3.3: List<Field> carRoute
loop
4: execute list in one second beat
5: trigger updateRoute after every beat
6: setNewDestinationField()
Powered By Visual Paradigm Community Edition

2.2 Use case and message description:

Title	Request car route
Description and Use Case	To start the travel method (in order the car to move on the map) the CDG microservice sends a request to the ARS microservice to get an route depending on the position and destination of the car. <u>Use case:</u> • After setting the position and destination of the new car object, the microservice requests a route from the position field to destination field. • This is done by putting a car object with the set fields into a message and sending it to the ARS microservice. • The microservice calculates the route and puts it inside the car object which is again put in a message and sent back to the requester. • After receiving the new car object with the route to the destination, the travel method for this object is triggered.

Transport Protocol Details	UDP, Client endpoint: /carRoute UDP, Server endpoint: /carRoute

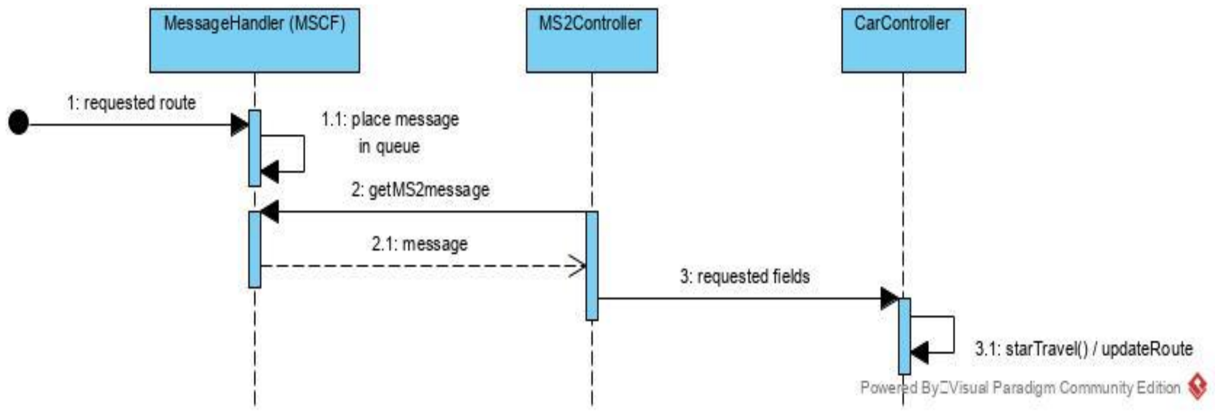
Sent Body Example (Client)	`{`
	`"MessageId": "23",`
	`"SourceIP": "10.101.101.13",`
	`"DestinationIP": "10.101.101.5",`
	`"DestinatedMS": "MS1/CarRoute",`
	`"Status": "null",`
	`"VisitedMS": "|MS2|",`
	`"TimeStamp": "2016-11-16 06:51:19.77",`
	`"Object": "Car",`
	`}`

Sent Body Description	▎MessageId: Message identifier which is unique and automatically assigned. The first number is always the number of the microservice. ▎SourceIP: The IP address of the microservice which generated the message. ▎DestinationIP: The IP address of the next hop microservice, or in best case the destination microservice. ▎DestinatedMS: Endpoint identifier for the destination microservice. ▎Status: Status of the executed process on the message in the current microservice. ▎VisitedMS:List of visited microservices till arrival at the destination. ▎TimeStamp: Date and time of the arrival at the microservice. ▎Object: The data which has to be processed at the destination.

Response Body Example (Server)	`{`
	`"MessageId": "15",`
	`"SourceIP": "10.101.101.5",`
	`"DestinationIP": "10.101.101.13",`
	`"DestinatedMS": "MS2/CarRoute",`
	`"Status": "null",`
	`"VisitedMS": "|MS1|",`
	`"TimeStamp": "2016-11-16 06:52:25.77",`
	`"Object": "Car",`
	`}`

Response Body Description	❙MessageId: Message identifier which is unique and automatically assigned. The first number is always the number of the microservice. ❙SourceIP: The IP address of the microservice which generated the message. ❙DestinationIP: The IP address of the next hop microservice, or in best case the destination microservice. ❙DestinatedMS: Endpoint identifier for the destination microservice. ❙Status: Status of the executed process on the message in the current microservice. ❙VisitedMS:List of visited microservices till arrival at the destination. ❙TimeStamp: Date and time of the arrival at the microservice. ❙Object: The data which has to be processed at the destination.
Error Cases	❙Invalid data type: Expected data type is Car in every other case exception is thrown. ❙Invalid Field type: Inside the route (List<Street>) there's a Field with a type undrivable.
Notes & Remarks	The Field class is further described in the MSCF Class Diagram and Description.

2.3 Sequence diagram response message:

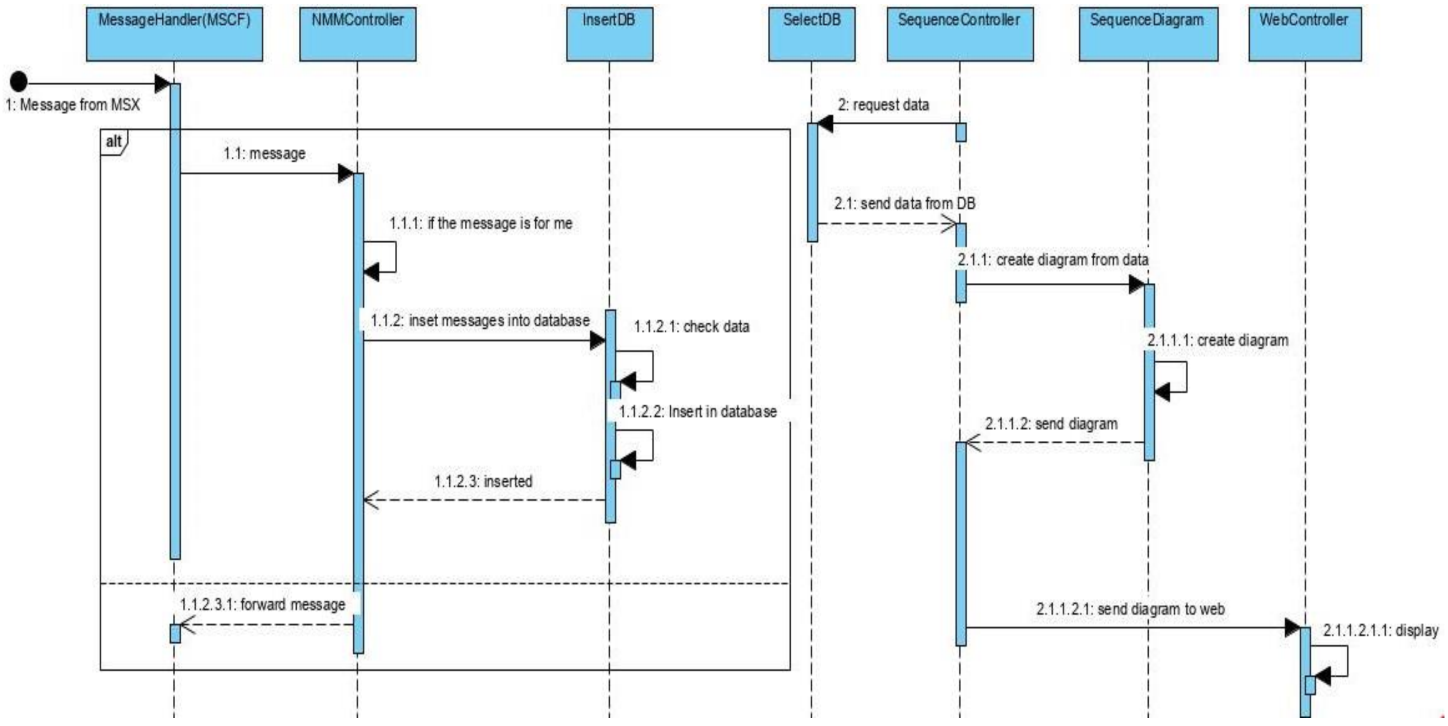


MS3 API Specification

(Network Monitoring and Maintenance Service):

MS3 is an informative microservice in this project. MS3 is the microservice whose main task is monitoring all information about messages that are exchanged between other microservices. The MS3 is communicating with other microservices to gather information inside own database or to forward the same to the other microservices. This microservice creates sequence diagrams from the message data that are previously stored and present those diagrams on the website. The website presents a visualization layer that communicates with the MS3 using Apache Tomcat that is predefined as part of the spring boot maven application.

Use cases and associated sequence diagrams



Title	Message list update
Description and Use Case	The microservice will receive a list of messages that were exchanged on multiple microservices. MS3 will save that data in its own database in multiple tables for each microservice. MS3 will use that data to make statistics that will be displayed on the web. Use Case: • Received data is saved in the database of MS3.

	• Data is fetched by "SequenceController" class. • Diagram is prepared for the publishing on the website. • Using JavaScript Mermaid JS the diagrams will be presented on the website.
Transport Protocol Details	UDP, endpoint: /list Rest (for Web presentation), endpoint: /diagrams

Sent Body Example (Client)	{
	    "MessageId": "23",
	    "SourceIP": "10.101.101.13",
	    "DestinationIP": "10.101.101.5",
	    "TypeOfMessage": "MS1/CarRoute",
	    "Status": "received",
	    "VisitedMS": "|MS2|",
	
	}

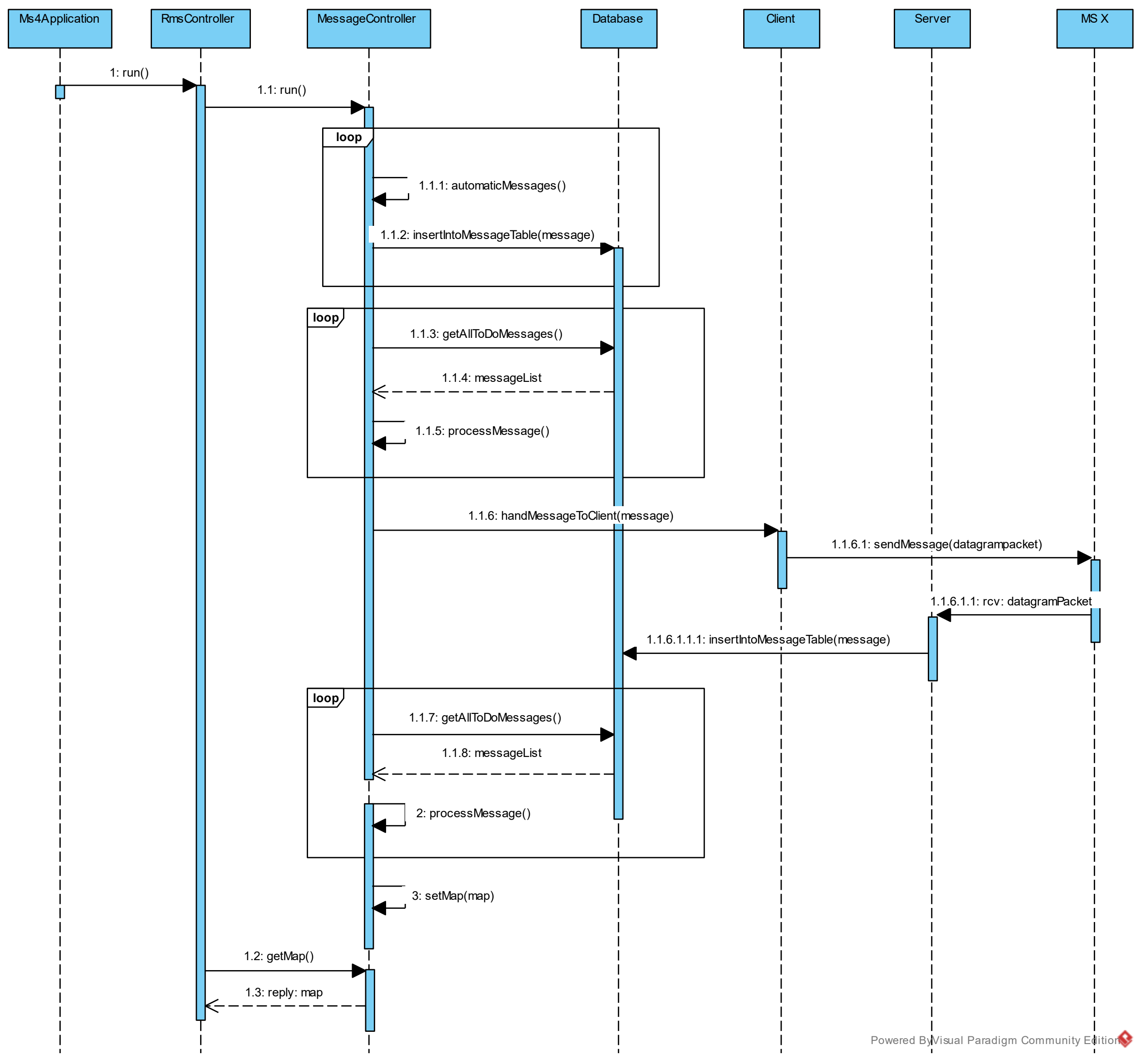
Sent Body Description	l MessageId: Message identifier which is unique and automatically assigned. The first number is always the number of the microservice. l SourceIP: The IP address of the microservice which generated the message. l DestinationIP: The IP address of the next hop microservice, or in best case the destination microservice. l TypeOfMessage: Description for the message that is exchanged between microservices. l Status: Status of the executed process on the message in the current microservice. l VisitedMS:List of visited microservices till arrival at the destination.
Response Body Example	Automatic framework(MSCF) replay response body.
Error Cases	l Invalid data type: Expected data type is List<Message> in every other case exception is thrown.
Notes & Remarks	

MS4 API Specification (Road Management Service):

MS4 represents an application that server like an interface between the CLI(User) and other Microservices. With it the user can have an overview of what is happening on the map, and to some extend control those by providing off-limit areas. The requests and responses with other microservices are done over Datagram Sockets which represent the mechanism for network communication via UDP. The communication with the CLI Application works over RESTful API, that uses HTTP requests to exchange data. All the logic for the package exchanging between Microservices is define in the MSCF that is imported by the micro- services. The following sequence diagrams and use cases represent the package exchange between the Road Management Service, the CLI Application and the others micro-services inside the network.

<u>Use cases and associated sequence diagrams</u>

1.1 Sequence-diagram Map request


Ms4Application
RmsController
MessageController
Database
Client
Server
MS X
1: run()
1.1: run()
loop
1.1.1: automaticMessages()
1.1.2: insertIntoMessageTable(message)
loop
1.1.3: getAllToDoMessages()
1.1.4: messageList
1.1.5: processMessage()
1.1.6: handMessageToClient(message)
1.1.6.1: sendMessage(datagrampacket)
1.1.6.1.1: rcv: datagramPacket
1.1.6.1.1.1: insertIntoMessageTable(message)
loop
1.1.7: getAllToDoMessages()
1.1.8: messageList
2: processMessage()
3: setMap(map)
1.2: getMap()
1.3: reply: map
Powered By Visual Paradigm Community Edition

Title	Map requested
Description and Use Case	The microservice MS4 sends a request to get a map object with a list of all available streets and their fields, so that the requester can work with those objects. <u>Use case - Send a map request to MS1:</u> • MS4 sends a request to MS1 to get a map from MS1 • After the response arrives MS4 will get the needed data from the updated Map • The data will be used to display the travel state of the cars or the areas that can be blocked or are already blocked, to unblock them
Transport Protocol Details	UDP, Client to MS1-endpoint: /map UDP, Server endpoint: /map

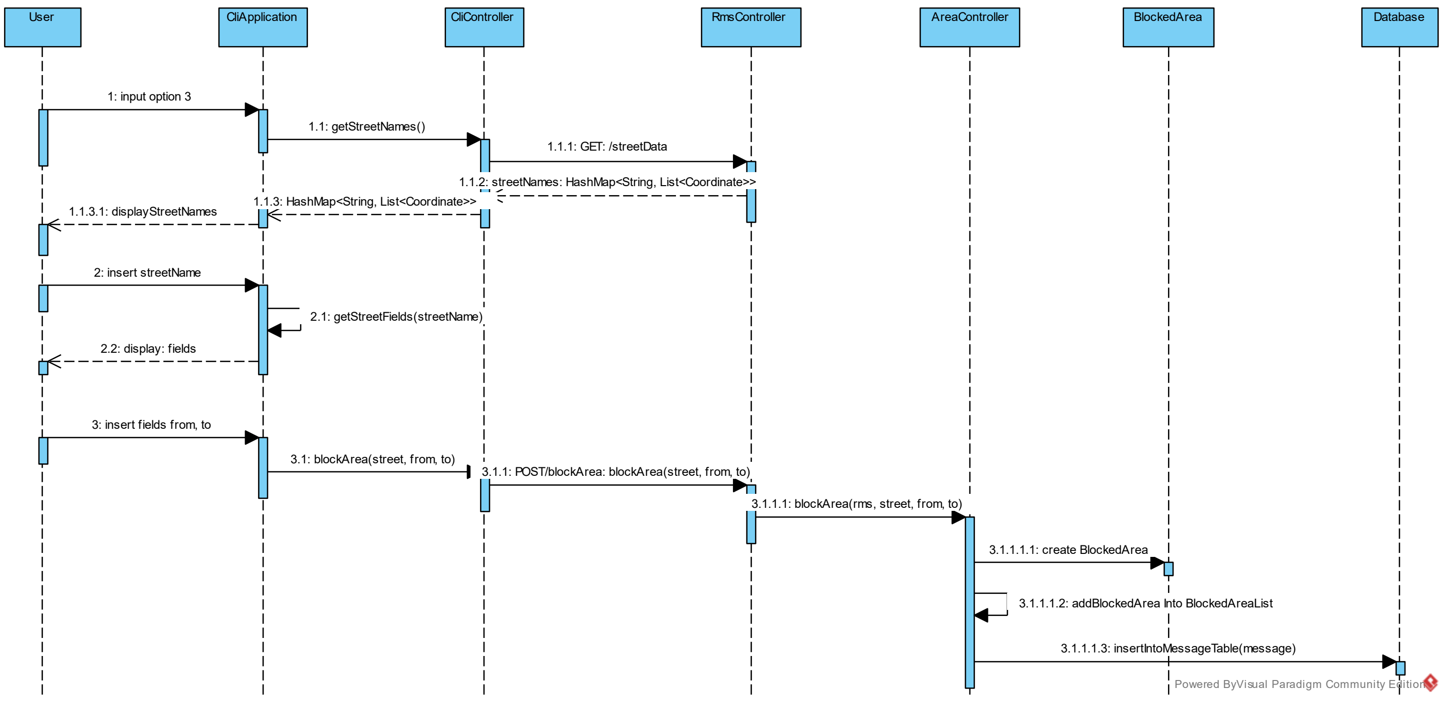
Sent Body Example	{
	    "MessageId": "41",
	    "SourceIP": "10.101.101.17",
	    "DestinationIP": "10.101.101.5",
	    "DestinatedMS": "MS1/map",
	    "Status": "null",
	    "VisitedMS": "|MS4|",
	    "TimeStamp": "2016-11-16 06:48:14.47",
	    "Object": "null",
	}

Sent Body Description	l MessageId: Message identifier which is unique and automatically assigned. The first number is always the number of the microservice. l SourceIP: The IP address of the microservice which generated the message. l DestinationIP: The IP address of the next hop microservice, or in best case the destination microservice. l DestinatedMS: Endpoint identifier for the destination microservice. l Status: Status of the executed process on the message in the current microservice. l VisitedMS:List of visited microservices till arrival at the destination. l TimeStamp: Date and time of the arrival at the microservice. l Object: Is empty cause no object needed for the request.

Response Body Example	`{`
	    `"MessageId": "12",`
	    `"SourceIP": "10.101.101.13",`
	    `"DestinationIP": "10.101.101.17",`
	    `"DestinatedMS": "MS4/map",`
	    `"Status": "null",`
	    `"VisitedMS": "|MS1|MS2|",`
	    `"TimeStamp": "2016-11-16 06:49:38.57",`
	    `"Object": "Map",`
	`}`

Response Body Description	- MessageId: Message identifier which is unique and automatically assigned. The first number is always the number of the microservice. - SourceIP: The IP address of the microservice which generated the message. - DestinationIP: The IP address of the next hop microservice, or in best case the destination microservice. - DestinatedMS: Endpoint identifier for the destination microservice. - Status: Status of the executed process on the message in the current microservice. - VisitedMS:List of visited microservices till arrival at the destination. - TimeStamp: Date and time of the arrival at the microservice. - Object: The data which has to be processed at the destination.
Error Cases	- Invalid endpoint: If an endpoint placed in DestinatedMS field does not match the predefined microservice endpoint, the message will be forwarded as long the time-to-live has not expired.
Notes & Remarks	The map object is further described in the part MSCF Description/Messages

1.2 Sequence-diagram Block Area


User
CliApplication
CliController
RmsController
AreaController
BlockedArea
Database
1: input option 3
1.1: getStreetNames()
1.1.1: GET: /streetData
1.1.2: streetNames: HashMap<String, List<Coordinate>>
1.1.3: HashMap<String, List<Coordinate>>
1.1.3.1: displayStreetNames
2: insert streetName
2.1: getStreetFields(streetName)
2.2: display: fields
3: insert fields from, to
3.1: blockArea(street, from, to)
3.1.1: POST/blockArea: blockArea(street, from, to)
3.1.1.1: blockArea(rms, street, from, to)
3.1.1.1.1: create BlockedArea
3.1.1.1.2: addBlockedArea Into BlockedAreaList
3.1.1.1.3: insertIntoMessageTable(message)
Powered ByVisual Paradigm Community Edition

Title	**Block Area**
Description and Use Case	**After the user inserted the data about the Area that he wants to block in the CLI, the MS4 sends the post request to MS1 so that he can, if possible, execute the block action.** **Use case:** • **Creates and sends a Street object, in which the specifics Fields are set to notDrivable, in a message object**
Transport Protocol Details	**REST, POST endpoint: /blockArea** **UDP, endpoint: /block**

Sent Body Example	`{`
	`    "streetName": "WähringerStraße",`
	`    "from": "2",`
	`    "to": "4"`
	`}`
	
	`{`
	`    "MessageId": "48",`
	`    "SourceIP": "10.101.101.17",`
	`    "DestinationIP": "10.101.101.5",`
	`    "DestinatedMS": "MS1/block",`
	`    "Status": "null",`

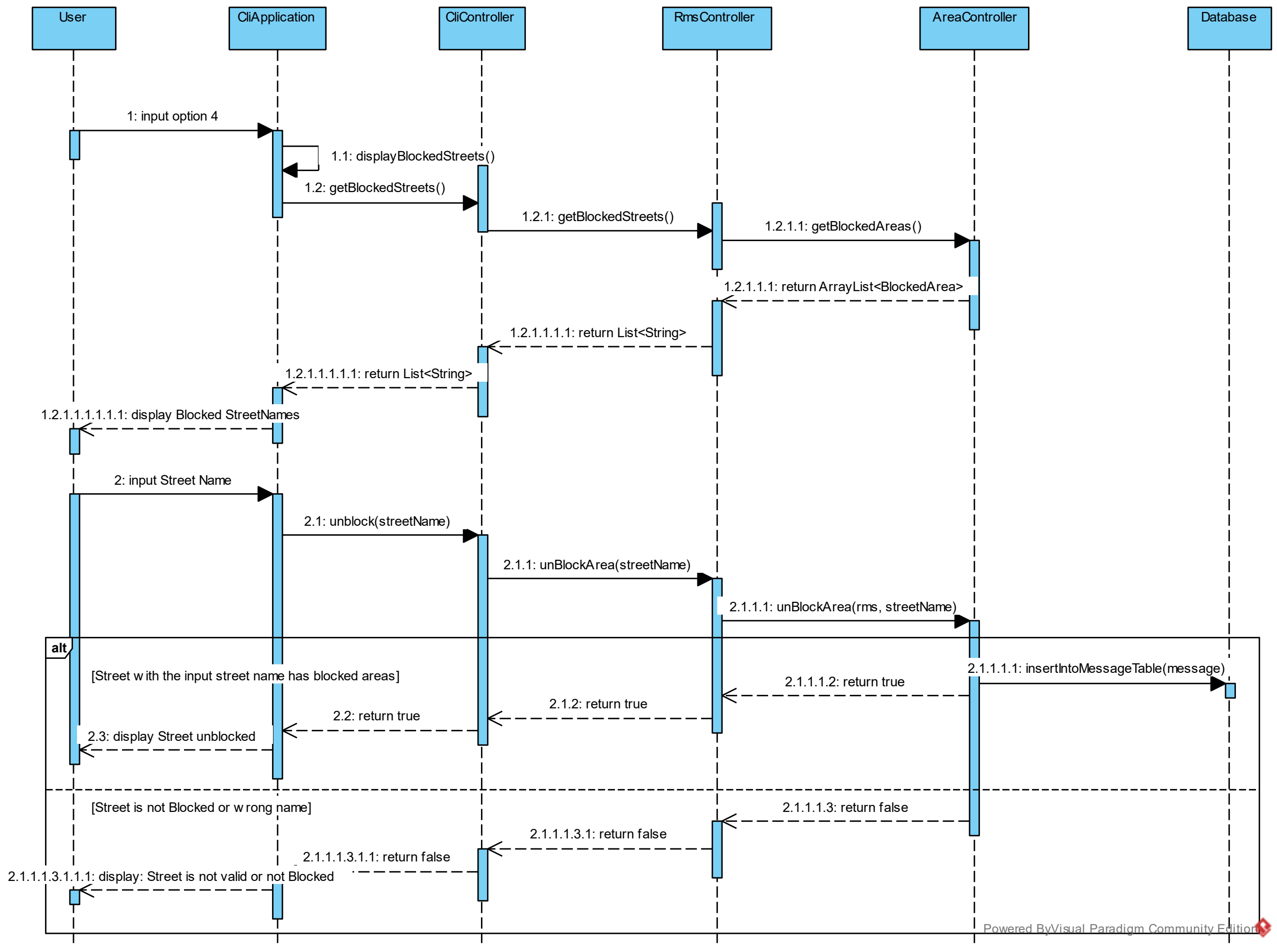
	"VisitedMS": "|MS4|",
	    "TimeStamp": "2016-11-16 06:50:19.77",
	    "Object": "Street",
	}

Sent Body Description	• streetName: Name of the street that is to be blocked • from: Field of the street from which the blocking starts • to: Field of the street where the blocking will end • MessageId: Message identifier which is unique and automatically assigned. The first number is always the number of the microservice. • SourceIP: The IP address of the microservice which generated the message. • DestinationIP: The IP address of the next hop microservice, or in best case the destination microservice. • DestinatedMS: Endpoint identifier for the destination microservice. • Status: Status of the executed process on the message in the current microservice. • VisitedMS:List of visited microservices till arrival at the destination. • TimeStamp: Date and time of the arrival at the microservice. • Object: The data which has to be processed at the destination.

Response Body Example	{
	    "blockState": "success"
	}
	
	{
	    "MessageId": "17",
	    "SourceIP": "10.101.101.5",
	    "DestinationIP": "10.101.101.17",
	    "DestinatedMS": "MS4/blockState",
	    "Status": "null",
	    "VisitedMS": "|MS1|",
	    "TimeStamp": "2016-11-16 06:50:28.77",
	    "Object": "Map",
	}

Response Body Description	- blockState: Message that is send back to the cli if the blocking was successful or not - MessageId: Message identifier which is unique and automatically assigned. The first number is always the number of the microservice. - SourceIP: The IP address of the microservice which generated the message. - DestinationIP: The IP address of the next hop microservice, or in best case the destination microservice. - DestinatedMS: Endpoint identifier for the destination microservice. - Status: Status of the executed process on the message in the current microservice. - VisitedMS:List of visited microservices till arrival at the destination. - TimeStamp: Date and time of the arrival at the microservice. - Object: The data which has to be processed at the destination.
Error Cases	- Received Area (Street and the Fields) from CLI is invalid: the StreetName and the field do not exist. - The area that is supposed to be blocked, has a car in it.
Notes & Remarks	The street object is further described in the part MSCF Description/Messages

1.3 Sequence-diagram Unblock Area


User
CliApplication
CliController
RmsController
AreaController
Database
1: input option 4
1.1: displayBlockedStreets()
1.2: getBlockedStreets()
1.2.1: getBlockedStreets()
1.2.1.1: getBlockedAreas()
1.2.1.1.1: return ArrayList<BlockedArea>
1.2.1.1.1.1: return List<String>
1.2.1.1.1.1.1: return List<String>
1.2.1.1.1.1.1.1: display Blocked StreetNames
2: input Street Name
2.1: unblock(streetName)
2.1.1: unBlockArea(streetName)
2.1.1.1: unBlockArea(rms, streetName)
alt
[Street with the input street name has blocked areas]
2.1.1.1.1: insertIntoMessageTable(message)
2.1.1.1.2: return true
2.1.2: return true
2.2: return true
2.3: display Street unblocked
[Street is not Blocked or wrong name]
2.1.1.1.3: return false
2.1.1.1.3.1: return false
2.1.1.1.3.1.1: return false
2.1.1.1.3.1.1.1: display: Street is not valid or not Blocked
Powered By Visual Paradigm Community Edition

Title	**Unblock Area**
Description and Use Case	**After the user inserted the data about the Area that he wants to unblock in the CLI, the MS4 sends the post request to MS1 so that he can, if possible, execute the unblock action.** **Use case:** • **Gets the street, provided from the CLI, and set the blocked Field to Drivable**
Transport Protocol Details	**REST, DELETE endpoint: /unblockArea** **UDP, endpoint: /unblock**

Sent Body Example	**Body from CLI to MS4:**
	
	{
	    "streetName": "Währingerstraße"
	}
	
	Body from MS4 to MS1:

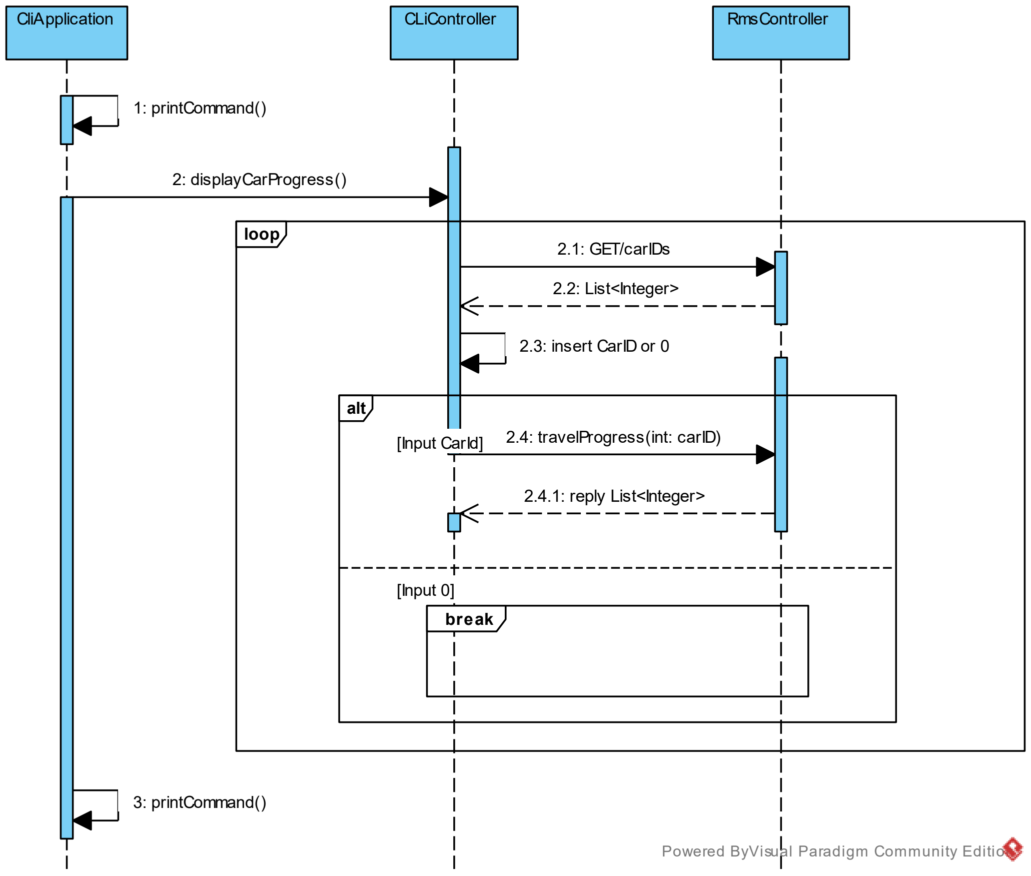
	`{`
	    `"MessageId": "410",`
	    `"SourceIP": "10.101.101.17",`
	    `"DestinationIP": "10.101.101.5",`
	    `"DestinatedMS": "MS1/unblock",`
	    `"Status": "null",`
	    `"VisitedMS": "|MS4|",`
	    `"TimeStamp": "2016-11-16 06:51:19.77",`
	    `"Object": "Street",`
	`}`

Sent Body Description	• streetName: Name of the street that is to be unblocked • MessageId: Message identifier which is unique and automatically assigned. The first number is always the number of the microservice. • SourceIP: The IP address of the microservice which generated the message. • DestinationIP: The IP address of the next hop microservice, or in best case the destination microservice. • DestinatedMS: Endpoint identifier for the destination microservice. • Status: Status of the executed process on the message in the current microservice. • VisitedMS:List of visited microservices till arrival at the destination. • TimeStamp: Date and time of the arrival at the microservice. • Object: The data which has to be processed at the destination.

Response Body Example	`{`
	    `"unblockState": "success"`
	`}`
	
	Body from MS1 to MS4: Updated Map
	
	`{`
	    `"MessageId": "111",`
	    `"SourceIP": "10.101.101.9",`
	    `"DestinationIP": "10.101.101.17",`
	    `"DestinatedMS": "MS4/unblockState",`
	    `"Status": "null",`
	    `"VisitedMS": "|MS1|MS3",`
	    `"TimeStamp": "2016-11-16 06:52:28.77",`
	    `"Object": "Map",`
	`}`

Response Body Description	❙ unblockState: Message that is send back to the cli if the unblocking was successful or not ❙ MessageId: Message identifier which is unique and automatically assigned. The first number is always the number of the microservice. ❙ SourceIP: The IP address of the microservice which generated the message. ❙ DestinationIP: The IP address of the next hop microservice, or in best case the destination microservice. ❙ DestinatedMS: Endpoint identifier for the destination microservice. ❙ Status: Status of the executed process on the message in the current microservice. ❙ VisitedMS:List of visited microservices till arrival at the destination. ❙ TimeStamp: Date and time of the arrival at the microservice. ❙ Object: The data which has to be processed at the destination.
Error Cases	❙ Received Area (Street) from CLI is invalid: the StreetName does not exist or Street is not blocked
Notes & Remarks	The street object is further described in the part MSCF Description/Messages

1.4 Sequence-diagram Car Progress



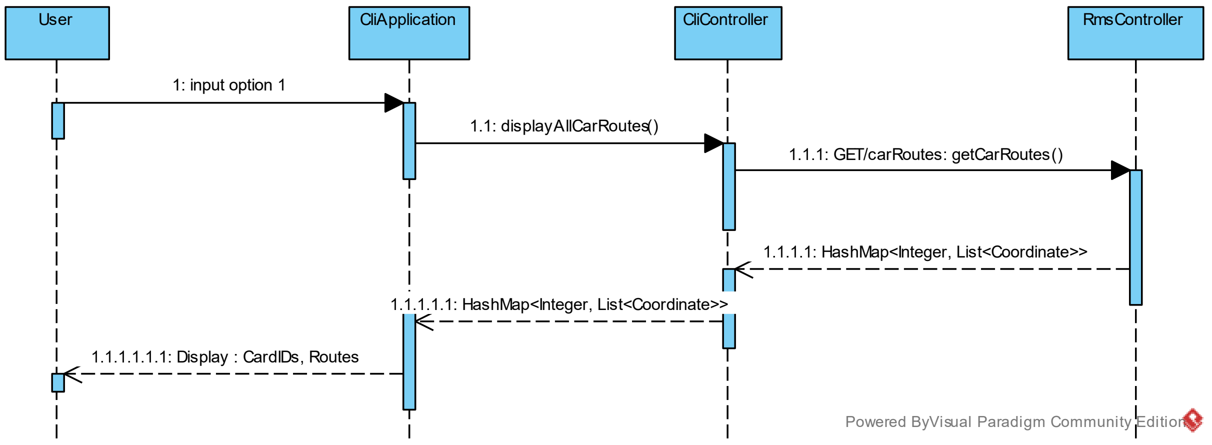
Title	Car Progress
Description and Use Case	After the user is provided with all the carIDs on the map, he can select one to see how far the car has travelled and still has to travel to his current destination Use case: • User choses the option to see the car progress and gets a list of carIDs • User inserts the id of the car he wants to see the progress • The number of traveled field and the number of the fields which the car has to travel to the current destination are displayed to the user in the CLI
Transport Protocol Details	REST, GET endpoint: /carIDs REST, GET endpoint: /progress/{id}

Body sent from cli	`{ "carId": "4" }`
Sent Body Description	• carId: ID of the chosen car

Response Body Example	Body from ms4 to cli for carIds request
	
	`{ "carIDs": "1,2,3,4" }`
	
	`{ " carId ": "111", " traveled ": "10.101.101.9", " toTravel ": "10.101.101.17", }`

Response Body Description	• carIDs: List of IDs of the cars on the map • carId: ID of the chosen car • traveled: Number of field traveled so far in the current Route • totravel: Number of field left until current Destination
Error Cases	• The Map data of the car is not still provided/arrived from ms1 • The inserted carID is not valid
Notes & Remarks	

1.5 Sequence-diagram Car Routes


User
CliApplication
CliController
RmsController
1: input option 1
1.1: displayAllCarRoutes()
1.1.1: GET/carRoutes: getCarRoutes()
1.1.1.1: HashMap<Integer, List<Coordinate>>
1.1.1.1.1: HashMap<Integer, List<Coordinate>>
1.1.1.1.1.1: Display : CardIDs, Routes
Powered By Visual Paradigm Community Edition

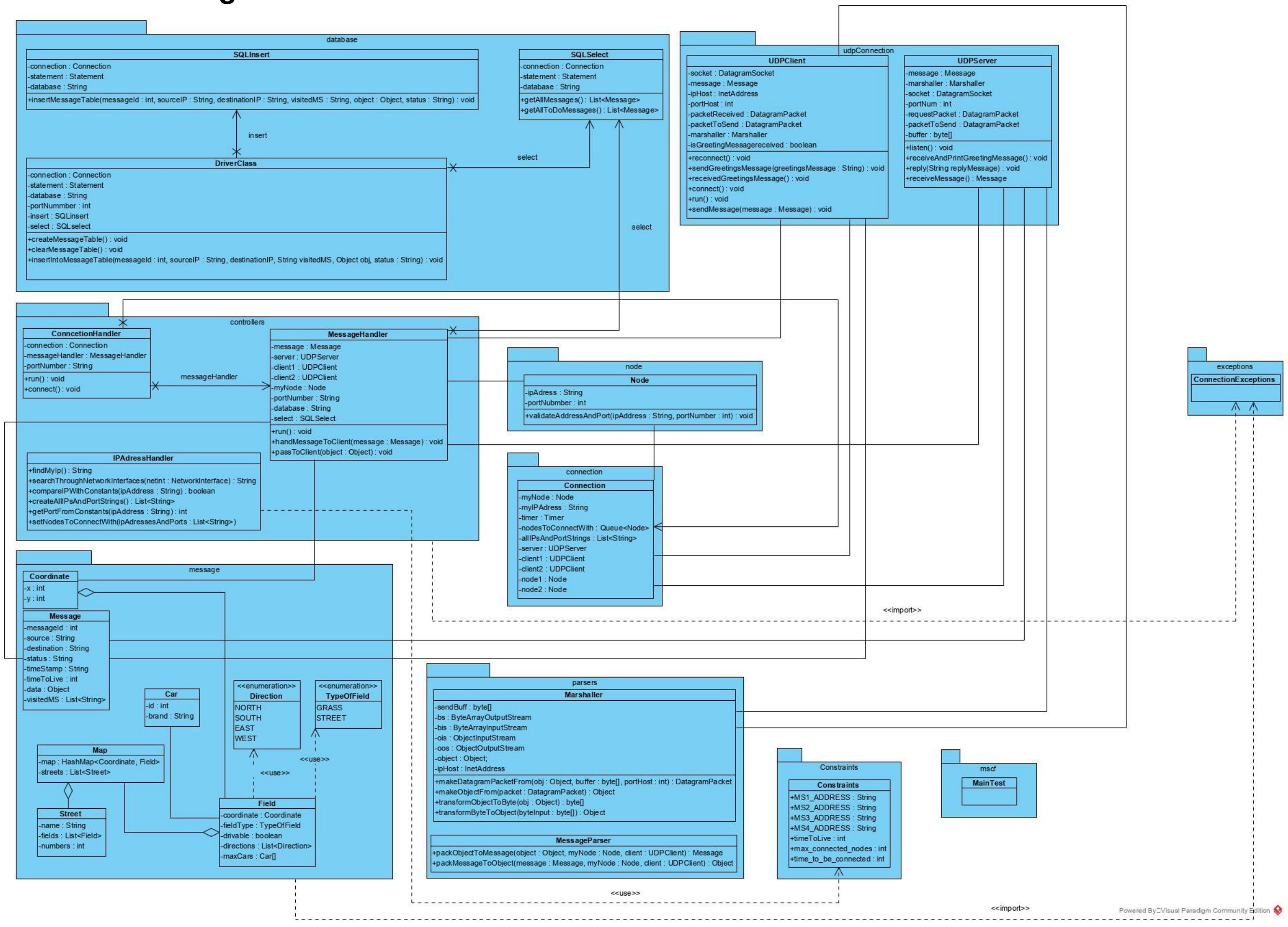
Title	**Car Routes**
Description and Use Case	**All cars and their Routes from current position to their destination will be displayed to the user in the CLI** **Use case:** • **User choses the option to see all car Routes** • **All cars and their routes will be displayed**
Transport Protocol Details	**REST, GET endpoint: /carRoutes**
Response Body Example	```{ "CarId": "1", "Coordinates": [{ "x": "0", "y": "1", "x": "0", "y": "2", }] "CarId": "2", "Coordinates": [{ "x": "5", "y": "4",```

	```
	        "x": "4",
	        "y": "4",

	      }

	    ]


	}
	```
Response Body Description	▎carId: ID of the car ▎Coordinates: Route will be presented over the coordinates of the fields that the car has to pass to get to the destination
Error Cases	▎ The Map data of the car is not still provided/arrived from ms1
Notes & Remarks	

Class diagrams (and skeleton-like[1] implementations):

MSCF Class Diagram and Description (Peer 2 Peer Communication Framework):

UML Class Diagram:


database
SQLInsert
-connection : Connection
-statement : Statement
-database : String
+insertMessageTable(messageId : int, sourceIP : String, destinationIP : String, visitedMS : String, object : Object, status : String) : void
SQLSelect
-connection : Connection
-statement : Statement
-database : String
+getAllMessages() : List<Message>
+getAllToDoMessages() : List<Message>
insert
select
DriverClass
-connection : Connection
-statement : Statement
-database : String
-portNummber : int
-insert : SQLInsert
-select : SQLselect
+createMessageTable() : void
+clearMessageTable() : void
+insertIntoMessageTable(messageId : int, sourceIP : String, destinationIP, String visitedMS, Object obj, status : String) : void
select
udpConnection
UDPClient
-socket : DatagramSocket
-message : Message
-ipHost : InetAddress
-portHost : int
-packetReceived : DatagramPacket
-packetToSend : DatagramPacket
-marshaller : Marshaller
-isGreetingMessagereceived : boolean
+reconnect() : void
+sendGreetingsMessage(greetingsMessage : String) : void
+receivedGreetingsMessage() : void
+connect() : void
+run() : void
+sendMessage(message : Message) : void
UDPServer
-message : Message
-marshaller : Marshaller
-socket : DatagramSocket
-portNum : int
-requestPacket : DatagramPacket
-packetToSend : DatagramPacket
-buffer : byte[]
+listen() : void
+receiveAndPrintGreetingMessage() : void
+reply(String replyMessage) : void
+receiveMessage() : Message
controllers
ConncetionHandler
-connection : Connection
-messageHandler : MessageHandler
-portNumber : String
+run() : void
+connect() : void
messageHandler
MessageHandler
-message : Message
-server : UDPServer
-client1 : UDPClient
-client2 : UDPClient
-myNode : Node
-portNumber : String
-database : String
-select : SQLSelect
+run() : void
+handMessageToClient(message : Message) : void
+passToClient(object : Object) : void
IPAdressHandler
+findMyIp() : String
+searchThroughNetworkInterface(netint : NetworkInterface) : String
+compareIPWithConstants(ipAddress : String) : boolean
+createAllIPsAndPortStrings() : List<String>
+getPortFromConstants(ipAddress : String) : int
+setNodesToConnectWith(ipAdressesAndPorts : List<String>)
node
Node
-ipAdress : String
-portNubmber : int
+validateAddressAndPort(ipAdress : String, portNumber : int) : void
connection
Connection
-myNode : Node
-myIPAdress : String
-timer : Timer
-nodesToConnectWith : Queue<Node>
-allIPsAndPortStrings : List<String>
-server : UDPServer
-client1 : UDPClient
-client2 : UDPClient
-node1 : Node
-node2 : Node
exceptions
ConnectionExceptions
<<import>>
message
Coordinate
-x : int
-y : int
Message
-messageId : int
-source : String
-destination : String
-status : String
-timeStamp : String
-timeToLive : int
-data : Object
-visitedMS : List<String>
Car
-id : int
-brand : String
<<enumeration>>
Direction
NORTH
SOUTH
EAST
WEST
<<enumeration>>
TypeOfField
GRASS
STREET
<<use>>
<<use>>
Map
-map : HashMap<Coordinate, Field>
-streets : List<Street>
Street
-name : String
-fields : List<Field>
-numbers : int
Field
-coordinate : Coordinate
-fieldType : TypeOfField
-drivable : boolean
-directions : List<Direction>
-maxCars : Car[]
parsers
Marshaller
-sendBuff : byte[]
-bs : ByteArrayOutputStream
-bis : ByteArrayInputStream
-ois : ObjectInputStream
-oos : ObjectOutputStream
-object : Object;
-ipHost : InetAddress
+makeDatagramPacketFrom(obj : Object, buffer : byte[], portHost : int) : DatagramPacket
+makeObjectFrom(packet : DatagramPacket) : Object
+transformObjectToByte(obj : Object) : byte[]
+transformByteToObject(byteInput : byte[]) : Object
MessageParser
+packObjectToMessage(object : Object, myNode : Node, client : UDPClient) : Message
+packMessageToObject(message : Message, myNode : Node, client : UDPClient) : Object
Constraints
Constraints
+MS1_ADDRESS : String
+MS2_ADDRESS : String
+MS3_ADDRESS : String
+MS4_ADDRESS : String
+timeToLive : int
+max_connected_nodes : int
+time_to_be_connected : int
mscf
MainTest
<<use>>
<<import>>
Powered By Visual Paradigm Community Edition

Description:

Constraints:

Constants: Holds four IPAddresses of team- members that MSs can take, timeToLive (e.g. 300miliseconds) time for listener UDPClient to wait for response, Max_connected_nodes- (represents number of at most two MSs connected with particular MS), time_to_be_connected (10seconds requirement before reconnection)

Parsers:

MessageParser: "(Un)Wrapper", (un)packs data to Message object to be exchanged between two MSs.

Marshaller: Role of this class is to convert Message object to and from DatagramPacket as a datatype which is exchanged. This class is also used to

---

1 This includes classes, interfaces, methods, return types, parameters and so on. The methods' body shall be empty.

transform Object to byte[ ] array in order to pack it as a “data” attribute with other attributes of a Message entity into the database and vice- versa.

Connection:
Connection: Holds data necessary for connection purposes- Node myNode(see node), Timer- instance of java’s Timer class to schedule the repetitive task, Queue<Node> nodesToConnectWith- collection with two node objects randomly picked to connect, List<String> allIPsAndPortStrings- all IPs and port number combinations, UDPServer server (see UDPServer), UDPClient client1, UDPClient client2 (see UDPClient). As it implements Runnable interface it does it’s work independently, in a run method it polls Node objects from collection, instantiates UDPClients and starts their threads.

Node:

Node: It holds IPAddress String and port number- int.

UDP_Connectors:

UDPServer: Opens DatagramSocket on a given port (int) an waits for a DatagramPacket. It needs Marshaller to unpack DatagramPacket and convert it to Message and to pack simple „OK“ as a response when a Message (or simple Greeting at the begining) is received. Instance of this class is intended to be open whole the time as MS runs and receives and sends back response. Received Messages are here put into the database for safety and statistical purposes.

UDPClient: There are two instances of this class in compliance with restriction- that a MS can be connected with to two MSs. It has properties almost same as UDPClient, except InetAddress- java’s class for address of the recipient and isGreetingMessagereceived boolean- because before exchanging data between MSs it must receive signal that it is connected with particular MS so the exchange can begin.

Exception:
ConnectionException: This class extends java’s Exception class has name String and message String. It is used to pack its’ name and message, if error occurs, into the Message „header“.

Database:

DriverClass: Connection connection- Instance of java’s interface to establish connection with database and create prepared statement. It creates database table for Message object and encapsulates methods from following two classes which it also instantiates and closes the connection with database.

SQLinsert: It inserts Message object into a database table with prepared statement.

SQLselect: This class hast wo methods: getAllToDoMessages() returns List<Message> whose „status“ attribute is null or „“ an empty String- messages with an unknown status and getAllMessages()- returns List<Message> of all messages in database.

Messages:
Message: This class is used as an envelope to wrap around an Object- real data that needs to be exchanged over network. It has serialVersionUID- because it implements Serializable interface (for storing into database as array of bytes and conversion to DatagramPacket), UUID corellationalID- unique ID, String source- IP and port of MS sender of this message, private String destination- in some cases it is IP of receiver, String status- Forwarded/Processed, String timeStamp- moment when the message is sent from MS, String endpoint- used to determine what to do with it when it reaches destination MS, Object data- real data wrapped inside o fit (see other classes of this package, visitedMS- List of MSs one message has visited along the way. It hast wo constructors- one for wrapping the data and other for statistical purposes.

Map: Represents a map of the city. It is going to be generated inside of MS1 each time it starts. For the assignment purposes it is intended to be kind of chess- boars 15x15. Properties of this class are HashMap<Coordinate, Field> map- collection of the Fields as value and Coordinate as key, as well as collection of Street objects - List<Street> streets.

Field: Represents a single unit, that map is comprised of. Each Field has Coordinate- position, TypeOfField fieldType- to distinguish i fit is drivable area or not, boolean drivable- flag used for blocking some areas if needed, List<Direction> directions- it can have at least two (every street is bidirectional) up to four (if the Field is at the crossroad), Car[ ]maxCars- holds Car objects that can be on it at the same time. It depends on directions (we assume that it is possible for two cars to pass by each other in the street and to be more than two cars at the crossroads at the same time).

Car: Represents a self- driving car object , passed by MS2 (CDG). It has id, brand String, List<Field> route Field- current location, Field destination and Direction – if two cars are on the same field, they must use different directions.

Coordinate: Represents position. Properties are int x and int y and validation() method.

TypeOfField: There are two field types- STREET and GRASS- Enum.

Direction: There are four different directions to move across the map- EAST, WEST, SOUTH, NORTH- Enum.

Street: This class is used to represent real- life street. It is List of Fields sorted by Coordinate, which makes blocking certain area easier.

Controlers:
IPAddressHandler: The functionality of finding an IPAddress of a MS in its' VPN is encapsulated in findMyIP()method. It iterates through its' NetworkInterfaces and finds IPAddress from the list. SetNodesToConnectWith() finds randomly two other IPAddress and ports for MS to connect.

MessageHandler: This class is used to pass Message objects between UDPServer and UDPClient as well as to hand them over to MS if they should be processed. It alse takes all messages from database for statistical analysis on MS3. The properties of this class are passed by ConnectionHandler after its' instantiation. Functionality of it is done in parallel with other classes, so it implements Runnable interface- in its' run() method currently is enabled to pass Message object to UDPClient to send it through network.

ConnectionHandler: Instantiates MessageHandler and Connection within its' constructor, takes a port number as an argument. It extends TimerTask java's class so the connection task can be repeated each 10 seconds. It starts MSCF by calling connect() method which starts UDPServer thread (currently only once when app runs and makes it listen for other MSs trying to connect) and schedules connection's timer to repeat connecting to randomly chosen two MS.

Improvement recommendations:

<u>*Response:*</u> It could be used to answer the request from particular MS and placed under the Messages package (MSCF) whether as constructor of Message class or a class itself. Additional properties: CorespondedMessageID UUID of the corresponded message, destination IP+Port from message source, exception name String, exception message String- Error handling, state OK/ERROR, Object data.

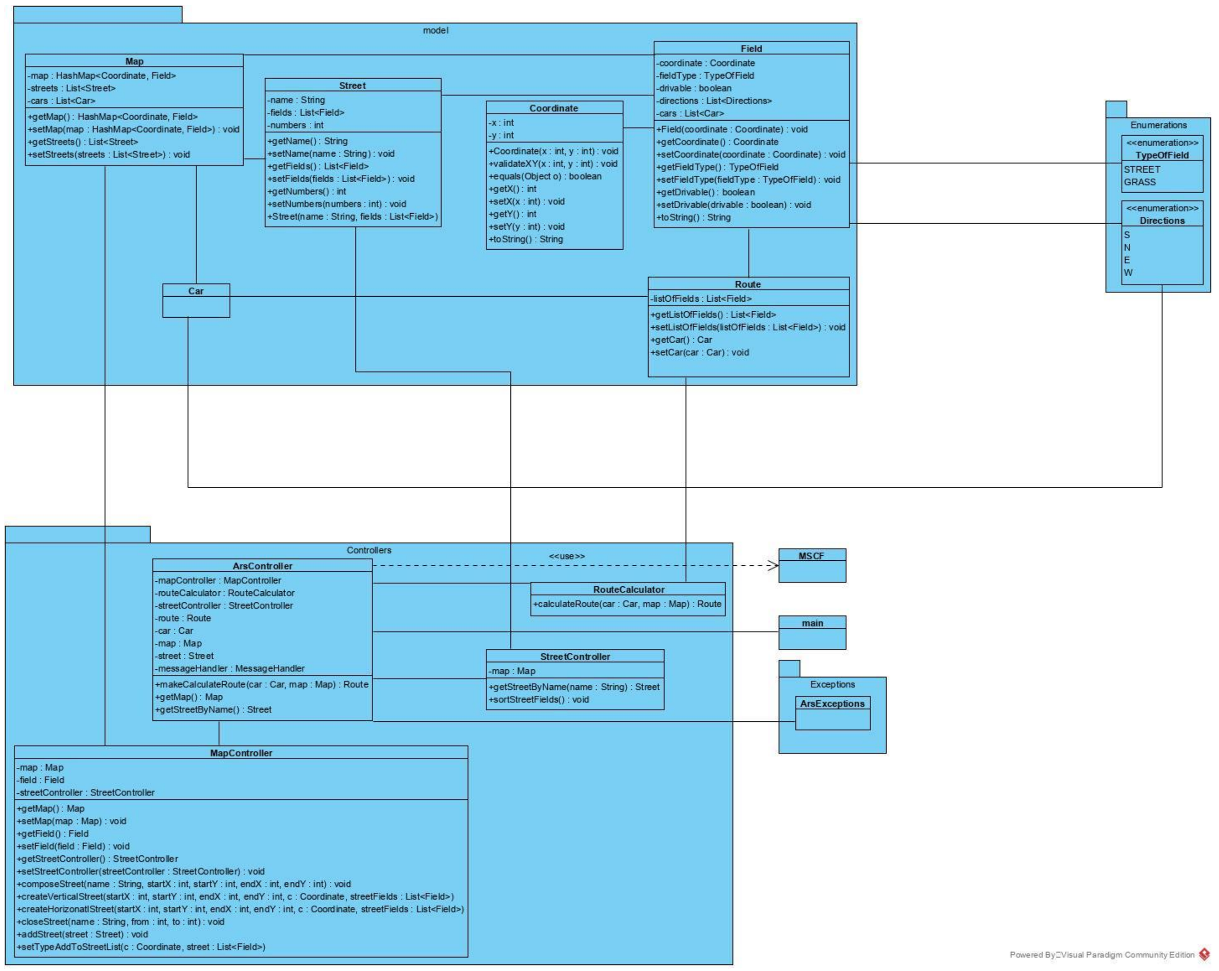
Connection:

<u>*UDPClient:*</u> Having in mind that MS2 should be able to run as a various number of instances (from 5 to 10) and other MSs only one for each (for the simplicity purposes) it could be allowed to have a constant range of Port numbers (from 3000 to 3015) in order to simplify the scanning process. Client part opens a DatagramSocket for each combination of given IP Address (String) and port number (other than it's own port number) and sends a simple greeting message (e.g. "Hello from MS.."), waits for a given period of time (e.g 300 milliseconds) for an answer from MS's Server part. As soon as answers from two different servers have arrived- this "greeting process" is over, Address- Port combinations are saved as Node- objects where the Messages will be sent for the rest of the 10 seconds interval. Greeting should be run simultaneously as a task by multiple workers, until the 2 Address and Port combinations are found.

<u>*UDPClient*</u>: Message to forward and requests/response from its' own MS Should send messages to previously determined (through "greeting" process) two MSs the rest of the 10 seconds. Each message sending is a task done concurrently by pool of workers (ExecutorService). After each greeting process. All messages with status "not sent" are taken from DB and placed in a list to be forwarded to connected MSs.

MS1 Class Diagram and Description (Automatic Routing Service):

UML Class Diagram


model
Map
-map : HashMap<Coordinate, Field>
-streets : List<Street>
-cars : List<Car>
+getMap() : HashMap<Coordinate, Field>
+setMap(map : HashMap<Coordinate, Field>) : void
+getStreets() : List<Street>
+setStreets(streets : List<Street>) : void
Street
-name : String
-fields : List<Field>
-numbers : int
+getName() : String
+setName(name : String) : void
+getFields() : List<Field>
+setFields(fields : List<Field>) : void
+getNumbers() : int
+setNumbers(numbers : int) : void
+Street(name : String, fields : List<Field>)
Coordinate
-x : int
-y : int
+Coordinate(x : int, y : int) : void
+validateXY(x : int, y : int) : void
+equals(Object o) : boolean
+getX() : int
+setX(x : int) : void
+getY() : int
+setY(y : int) : void
+toString() : String
Field
-coordinate : Coordinate
-fieldType : TypeOfField
-drivable : boolean
-directions : List<Directions>
-cars : List<Car>
+Field(coordinate : Coordinate) : void
+getCoordinate() : Coordinate
+setCoordinate(coordinate : Coordinate) : void
+getFieldType() : TypeOfField
+setFieldType(fieldType : TypeOfField) : void
+getDrivable() : boolean
+setDrivable(drivable : boolean) : void
+toString() : String
Car
Route
-listOfFields : List<Field>
+getListOfFields() : List<Field>
+setListOfFields(listOfFields : List<Field>) : void
+getCar() : Car
+setCar(car : Car) : void
Enumerations
<<enumeration>>
TypeOfField
STREET
GRASS
<<enumeration>>
Directions
S
N
E
W
Controllers
<<use>>
ArsController
-mapController : MapController
-routeCalculator : RouteCalculator
-streetController : StreetController
-route : Route
-car : Car
-map : Map
-street : Street
-messageHandler : MessageHandler
+makeCalculateRoute(car : Car, map : Map) : Route
+getMap() : Map
+getStreetByName() : Street
RouteCalculator
+calculateRoute(car : Car, map : Map) : Route
StreetController
-map : Map
+getStreetByName(name : String) : Street
+sortStreetFields() : void
MapController
-map : Map
-field : Field
-streetController : StreetController
+getMap() : Map
+setMap(map : Map) : void
+getField() : Field
+setField(field : Field) : void
+getStreetController() : StreetController
+setStreetController(streetController : StreetController) : void
+composeStreet(name : String, startX : int, startY : int, endX : int, endY : int) : void
+createVerticalStreet(startX : int, startY : int, endX : int, endY : int, c : Coordinate, streetFields : List<Field>)
+createHorizonatlStreet(startX : int, startY : int, endX : int, endY : int, c : Coordinate, streetFields : List<Field>)
+closeStreet(name : String, from : int, to : int) : void
+addStreet(street : Street) : void
+setTypeAddToStreetList(c : Coordinate, street : List<Field>)
MSCF
main
Exceptions
ArsExceptions
Powered By Visual Paradigm Community Edition

Description

Model:

Route: Represents the path from starting to destination Field for a particular Car. It Holds it as a list of Fields.

Car: Similar to MSCF object, it has a Route which is updated when requested and sent back to MS2(CDG).

Field: Similar to MSCF. Should contain cost (int) property which is used for Route calculation (if there is a car on the field and moves in the same direction like a car that needs route (re)calculated, it increases its' cost by one).

Controller:

MapController: Generates map when MS1 runs and creates Streets by calling

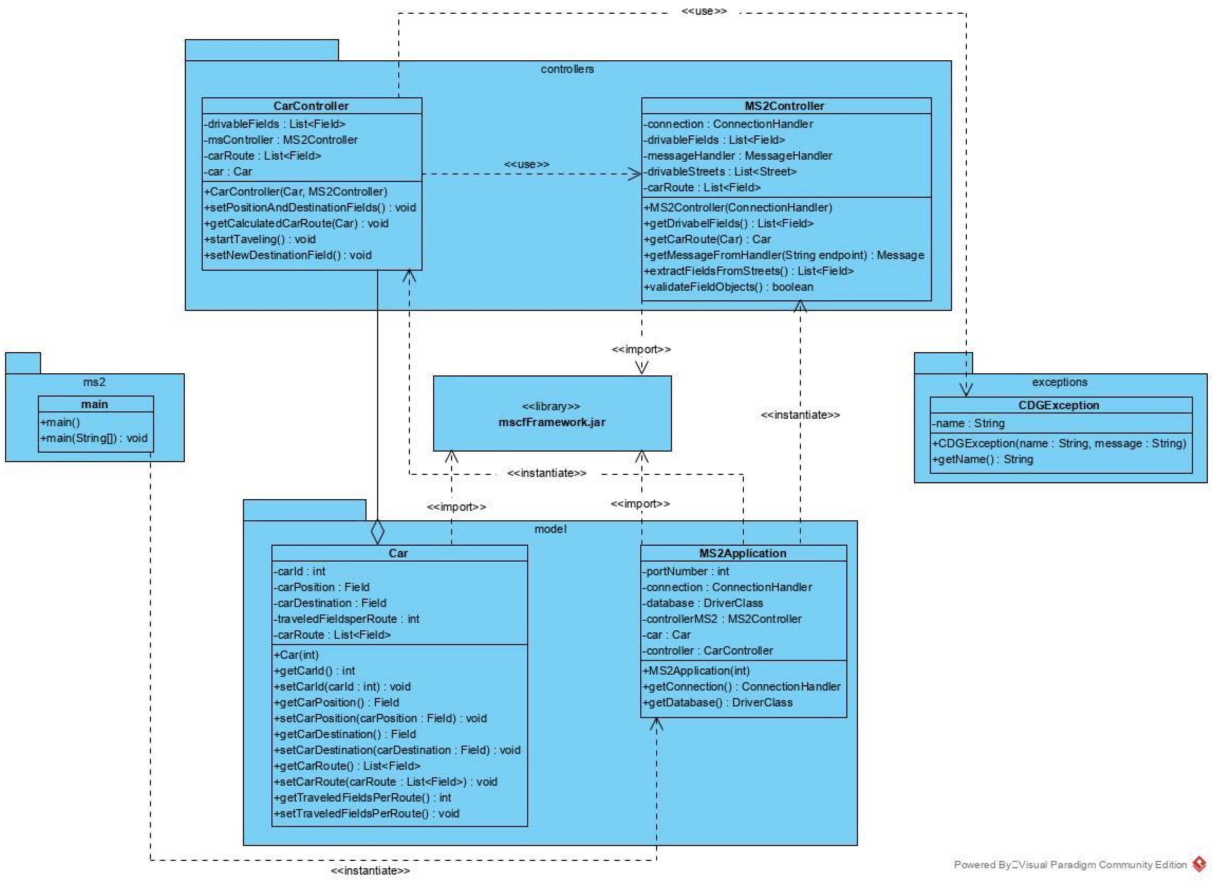
composeStreet(Street Name, int startX, int startY, int endX, int endY) multiple times (hardcoded). It also returns map, streets and blocks an area on the
map when requested.
ArsController: Main controller in the MS1. It communicates with Other controllers to pass (and get from) them required object (map, car, streets) as well as with the MessageHandler which carries them back to MSCF for sending to other MSs.
RouteController: Holds logic for calculation of the route for particular car. It takes its' current position, destination and uses Breadth First Search combined with Dijkstra algorithm to determine the fastest way to destination.

Exception

ArsException: Custom exception thrown to locate unwanted behaviour.

MS2 Class Diagram and Description (Car Data Gateway):

UML Class Diagram:



<<use>>
controllers
CarController
-drivableFields : List<Field>
-msController : MS2Controller
-carRoute : List<Field>
-car : Car
+CarController(Car, MS2Controller)
+setPositionAndDestinationFields() : void
+getCalculatedCarRoute(Car) : void
+startTaveling() : void
+setNewDestinationField() : void
<<use>>
MS2Controller
-connection : ConnectionHandler
-drivableFields : List<Field>
-messageHandler : MessageHandler
-drivableStreets : List<Street>
-carRoute : List<Field>
+MS2Controller(ConnectionHandler)
+getDrivabelFields() : List<Field>
+getCarRoute(Car) : Car
+getMessageFromHandler(String endpoint) : Message
+extractFieldsFromStreets() : List<Field>
+validateFieldObjects() : boolean
<<import>>
ms2
main
+main()
+main(String[]) : void
<<library>>
mscfFramework.jar
<<instantiate>>
exceptions
CDGException
-name : String
+CDGException(name : String, message : String)
+getName() : String
<<instantiate>>
<<import>>
<<import>>
model
Car
-carId : int
-carPosition : Field
-carDestination : Field
-traveledFieldsperRoute : int
-carRoute : List<Field>
+Car(int)
+getCarId() : int
+setCarId(carId : int) : void
+getCarPosition() : Field
+setCarPosition(carPosition : Field) : void
+getCarDestination() : Field
+setCarDestination(carDestination : Field) : void
+getCarRoute() : List<Field>
+setCarRoute(carRoute : List<Field>) : void
+getTraveledFieldsPerRoute() : int
+setTraveledFieldsPerRoute() : void
MS2Application
-portNumber : int
-connection : ConnectionHandler
-database : DriverClass
-controllerMS2 : MS2Controller
-car : Car
-controller : CarController
+MS2Application(int)
+getConnection() : ConnectionHandler
+getDatabase() : DriverClass
<<instantiate>>
Powered By Visual Paradigm Community Edition

Description:

mscfFramework: provides predefined classes and functions that are used for the network communication interaction with the database. Duplicate and redundant code is avoided because all of the microservices implement the framework foundation.

Controllers:

MS2Controller: The responsibility of this class is to forward the requests, handed over by the ClassController, to the microservice target, as well as to receive messages from other microservices, unpack them and transfer them to the microservice for processing. This is done over the classes implemented from the framework and local methods.

CarController: Controls the car object by updating its position, destination and route. Depending on the controlled process, transfers requests to the MS2Controller, to get a new car route from the ARS from example.

Model:

Car: It represents a set of properties and methods that are common to all objects of the car type. After starting the microservice the Constructor is used to initialize a new object with variables that provide the state of the class and its object.

MS2Application: Creates a series of new objects, used in the microservice to represent the car object travel and communicate with all other microservices in the network.

Exceptions:

CDGException: A custom exception class is created and used, so the user can now know what the exact exception is. It provide us with the flexibility to add attributes and methods that are not part of a standard Java exception. In these we stored additional information, like an application-specific error code.

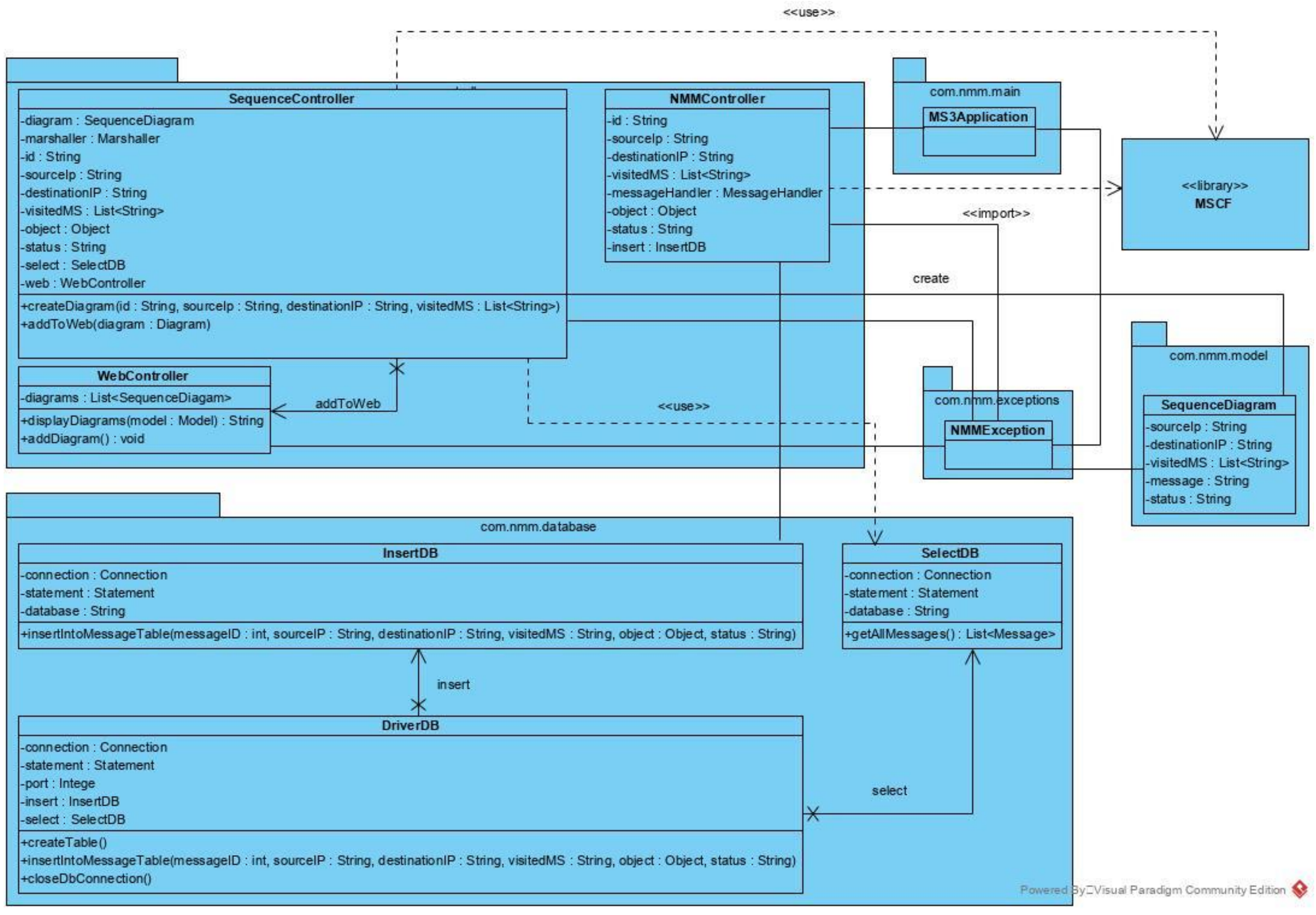
ms2:

main: The main class creates 5 instances of the CDG microservice, were every instance has her own database, messageHandler, car object and other logic provided by the CDG Application and mscf Framework.

MS3 Class Diagram and Description

(Network Monitoring and Maintenance Service):

UML Class Diagram :



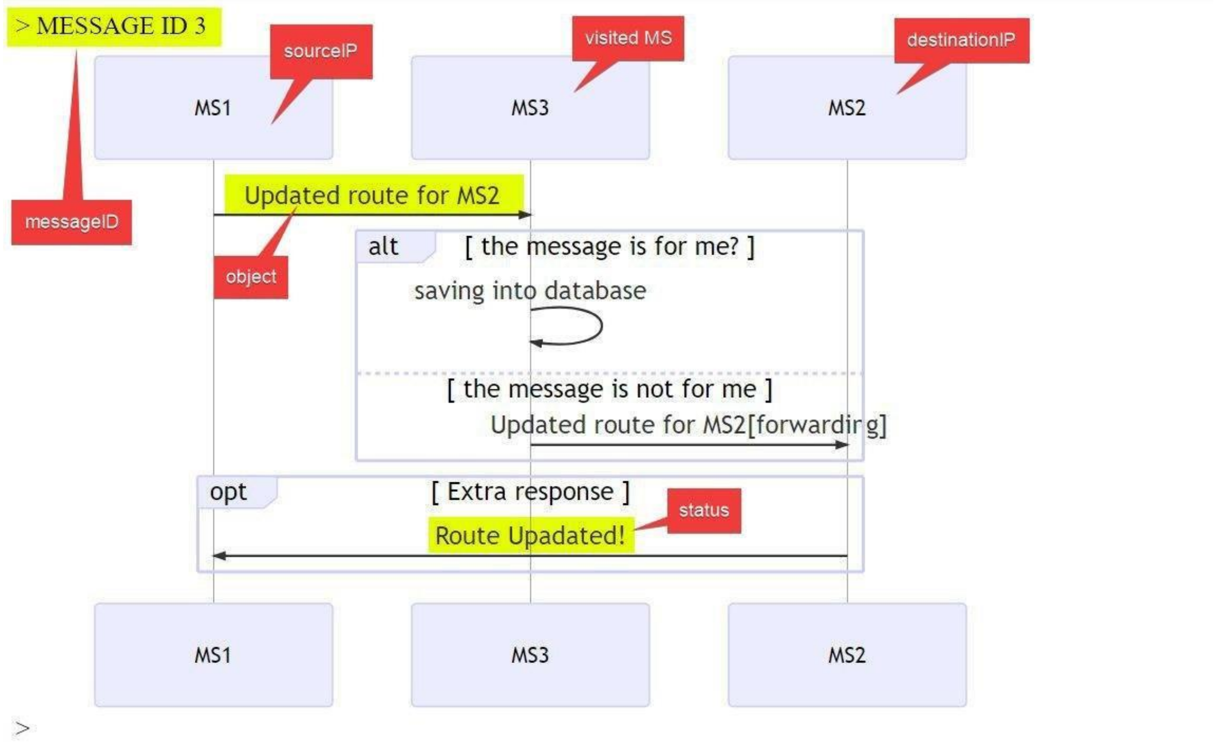
Description:

Controllers:

NMMController : The most significant controller of them all in this microservice. The main task of this controller is to use the advantages of the MSCF framework to realize where the messages should go. It got 2 options, to put received message in its own database if the message is maintained to this MS, or to forward the message to the wanted destination. The controller will unwrap the message, prepare the wanted results that will be presented for the database, and with functions of "InsertDB" will put those data in the database table. NMMController communicates with other controllers to pass them demanded messages. It is the first obligation in this microservice that will allow further evaluation of the tasks and information that this microservice should present.

SequenceController: SequenceController is the author of the diagrams that will be presented on the webpage of the application that this microservice provides. The main task and idea are to use data from the database with the help of the “SelectDB” class. Pull the information from the database, and then convert all of the data to a string, to be able to make a sequence diagram object. The bigger part of the data are already strings, but the object type isn't. Knowing the type of object, this controller should make a compatible diagram, it means that the diagram will represent what type of communication is shifted between those microservices. Those data need to be strings because we are using the mermaid JS library to represent those diagrams, and the most simple way is to use strings to interpret data in the HTML file.

WebController : The WebController is the administrator of diagrams, this controller collects all produced diagrams and presents them in nice shape with the help of mermaid JS library. This is the rest controller in this Spring boot application that uses the spring boot "model" to transfer wanted data to the HTML file. Of course, that would not be able with the Thymeleaf, server Java template generator for the web scenes. WebController attaches the name of the function and the diagram as a parameter in the model object and returns the string name of the HTML file where the diagram will be displayed. The application will run a localhost on port 8080 with the different endpoints for displaying data.



Html, presentation of sequence diagram object using "mermaid" JS library.

Database:

DriverDB : The class has the main task that probably has the same functionality that is in the practice of the mscf framework database. An occurrence to build a connection with a database and generate a qualified declaration. The DriverDB class produces a table for a Message where the data will be located and used for future composing of the sequence diagrams and statistics that will be presented to the webpage of this microservice.

InsertDB : This class is a simple solution for inserting the requested data into the message table. It inserts the data into a database table with an adapted SQL statement. This class will be practically used in the most important stage of the microservice and that is in NMMController. After checking all of the messages, reliable data will be inserted into the database with the help of the method that this class provides.

SelectDB: This class has a task to pull the data from the table in SequenceController to be able to build a properly and accurately sequence diagram using the method "getAllMessages()". Those data will be distributed in the SequenceDiagram class for further usage.

Model:

SequenceDiagram : This is the only class in this package. The name of the class already saying that this class is used for making sequence diagram objects. This class holds all valid information and data from the database that is coming from the SequecnceController. The main things that this class needs to have are the message-id that will be presented as a title of the diagram, source IP, destination IP and all visited microservices that will be converted to the name of the microservices, an object will be converted to the type of message that is traveling between microservices and status will decide the look of the diagram in the way if the message is dropped or successfully received. Look for an example in the WebController description.

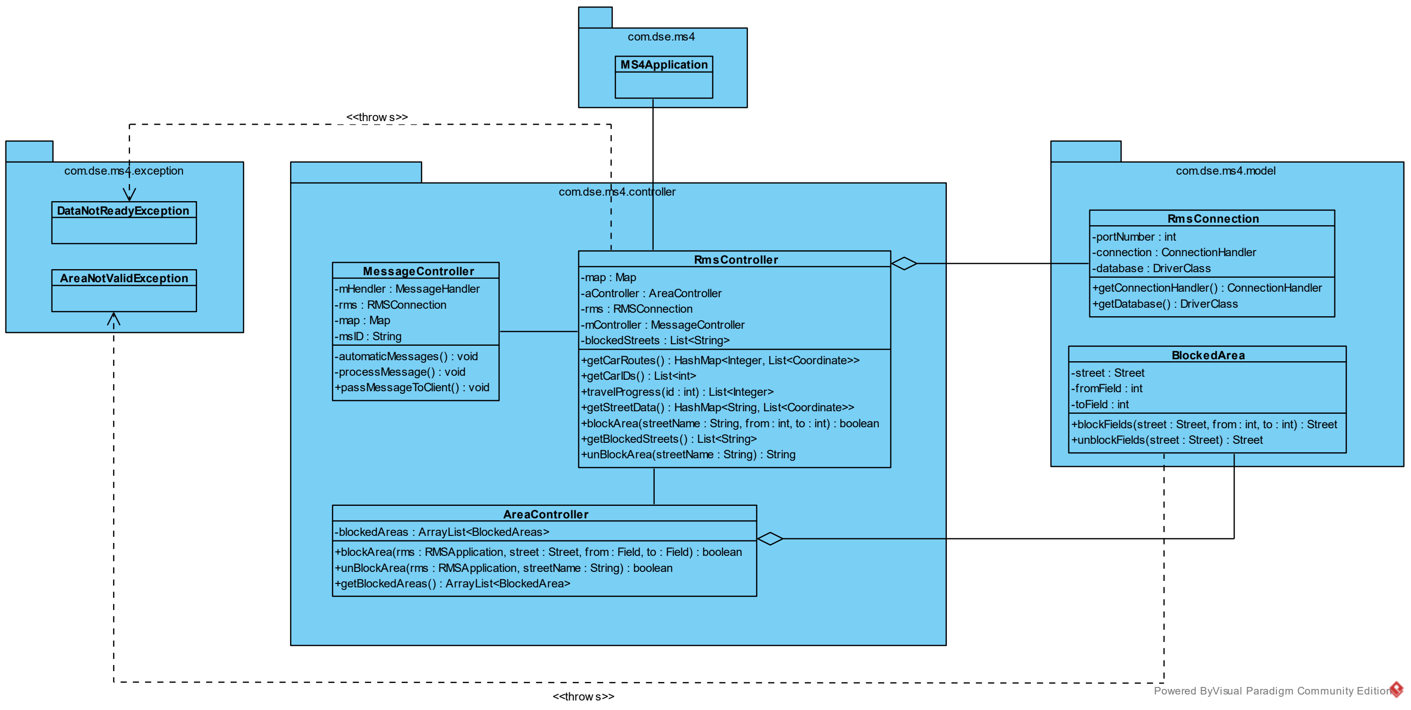
Exceptions:

NMMException: A custom exception class is designed to help us to exactly know where the problem is located in the code. There are a lot of possibilities for errors, especially in our project with network communication. This class will provide us with debugging and equips us with the information that will be generated through the methods of a standard Java exception. The main task is to be able to test and generate a clear workspace in the application.

Main:
*MS3Application :*This is the main class of the spring boot project. The main task is to initialize all necessary data and to show the workflow behind the curtains while the application is running.

MS4 Class Diagram and Description (Road Management Service):

UML Class Diagram:


com.dse.ms4
MS4Application
<<throws>>
com.dse.ms4.exception
DataNotReadyException
AreaNotValidException
com.dse.ms4.controller
MessageController
-mHendler : MessageHandler
-rms : RMSConnection
-map : Map
-msID : String
-automaticMessages() : void
-processMessage() : void
+passMessageToClient() : void
RmsController
-map : Map
-aController : AreaController
-rms : RMSConnection
-mController : MessageController
-blockedStreets : List<String>
+getCarRoutes() : HashMap<Integer, List<Coordinate>>
+getCarIDs() : List<int>
+travelProgress(id : int) : List<Integer>
+getStreetData() : HashMap<String, List<Coordinate>>
+blockArea(streetName : String, from : int, to : int) : boolean
+getBlockedStreets() : List<String>
+unBlockArea(streetName : String) : String
AreaController
-blockedAreas : ArrayList<BlockedAreas>
+blockArea(rms : RMSApplication, street : Street, from : Field, to : Field) : boolean
+unBlockArea(rms : RMSApplication, streetName : String) : boolean
+getBlockedAreas() : ArrayList<BlockedArea>
com.dse.ms4.model
RmsConnection
-portNumber : int
-connection : ConnectionHandler
-database : DriverClass
+getConnectionHandler() : ConnectionHandler
+getDatabase() : DriverClass
BlockedArea
-street : Street
-fromField : int
-toField : int
+blockFields(street : Street, from : int, to : int) : Street
+unblockFields(street : Street) : Street
<<throws>>
Powered By Visual Paradigm Community Edition

Description:

Model:

BlockedArea – Contains the Street, which is supposed to be blocked and the numbers of fields “from” and “to”, between which the field will be set as undrivable.

RmsConnection – Over this class the connection with the other Microservices will be established.

Controllers:

RmsController - this class is the main controller of MS4 which communicates with other controllers to pass and get required object and represents the connection between MS4(RMS) and the CLI Application. The communication is done with RESTfull Api. Depending on command from the CLI Application, the class can initiate the process to block an area, unblock an area, get specific data about Cars and their Routes and send it back to the CLI Application.

MessageController – the main task of this class is the handling of messages. After pulling the messages from the database, the messages will be individually processed. Depending on the message destination they can be forwarded or the data from the message will be extracted if the message was for MS4.

AreaController – this class provides the functionality to create BlockedAreas and unblock them if wanted. It also has a small overview over blocked areas as a List.

Exceptions:

DataNotReadyException – is a custom exception which is thrown if the necessary data for the cli is not still provided.

AreaNotValidException – is a custom exception which is thrown if the User, over the CLI, wants to block a street which does not exist, inputs number of field that the street doesn't have or one or more field have cars on it. The exception will also be thrown if the user tries to unblock an area that is not blocked.

Document the status of each Microservice and Framework Components :

MSCF (Peer 2 Peer communication framework):

- **Design decision**: By creating this communication framework we are implementing some basic principles characteristic for communication over network. Object classes are going to be created immutable (without setters) because we tend not to allow any possible influence on the data from outside. Wrapper pattern- to wrap the data into an envelope. Statistic messages are separated from ordinary data-messages, Router pattern will be used to decide to which MS a Message will be sent (if possible), Aggregator will be used in premises of MS3 to gather all Messages for statistical analysis.

- **Integration:** MSCF is framework which covers the connection and communication functionality. It is planned to be imported as a .jar file by every single MS. Connection (picking two randomly chosen MSs to connect with) and communication (Message exchange) happens “under the hood” and doesn’t need to be re- implemented. By simple instantiation of the ConnectionHandler and calling the connect() method- it will be started.

- **Testing and presentation:** Testing of the single methods is going to be done by using Junit5 Framework for unit- testing. Connection and sending basic messages are done in a manual manner- at least two team members must be involved to try different scenarios and discuss whether the outcome is as desired or not. In some cases, single components are built as a separate project to test the functionality. In situations when we must test certain functionalities alone, without other team members, as mentioned- either place functionalities in separate projects and simulate communication or connecting between entity with “localhost”(and given port number) and given vpn address, or by starting at least two MS on the same machine with different port numbers.
- **Status:**

Implemented:

- Sending greeting by UDPClient/ response from UDPServer (must be integrated with next point)
- Sending simple Message object UDPClient/ response from UDPServer(must be integrated with previous point)
- Passing the Message from MessageHandler to UDPClient (but only from given list of hardcoded Messages)
- Inserting Message with Object data to database
- Selecting Messages from database
- Finding own IPAddress and choosing randomly MSs to connect to(approach must be changed due to miss-understanding the requirements)

Not implemented:

- Take over Message from UDPServer by MessageHandler
- Decision what to do with it (Process/Forward)
- Pass the Message to MSx for processing
- Handle Error cases

MS1 (Automatic Routing Service):

- **Design decision:** This MS is intended to be created by using basic MVC principles, although the we are going to omit visual representation. Data is separated to the model classes and the behaviour is handed to controllers for each class. Map should be singleton, because it should be instantiated only once.

- **Deployment:** This MS is going to be run- either by exporting it to .jar file, compilation and run through CLI (with passing the port number as an argument) or by importing (the latest version from git repository) into Eclipse IDE (2019-12) and setting port number in the "Run configuration/ Arguments". Due the fact that its' work is going to be done automatically without interference of end- user it can't be reached in that manner. IPAddress will be read from NetworkInterfaces of the machine it works on (part of MSCF) and the port number will be given. The Messages are going to reach it either through "destinatedMS"/"endpoint" property inside of Message object or its' IPAddress (response- on error- will have an IP of a sender set as destination IP).

- **Testing and presentation:** Single methods are going to be with Junit5 framework as unit tests tested. After finalization of single functionalities, integration will be tested either with other team members, or (as explained in MSCF) with combinations "localhost" and port and given vpn to simulate communication and connecting.

- **Status:**

Implemented:
- Map generating
- Streets generating and sorting fields
- Blocking the area

Not implemented:
- Unit tests
- Route calculation- BFS (Breadth First Search + Dijkstra)
- Create Map as singleton
- Passing the Object to MessageHandler
- Set the direction to different fields
- Involve cost as a property of the field and logic behind it for route calculation

MS2 (Car Data Gateway):

- ***Design decisions:***

The microservice is divided according to functionality packages. This means that data and the controllers for that data are stored in separate packets. In addition, a microservice-specific exception class has been created to make it easier to identify errors in the process.

The automation of the whole process made it very easy to operate the microservice, even for third parties. Depending on how many processes (CDGs) the user wants to initialize, the run button simply has to be pressed that often. A maximum of 15 processes can run at the same time because there are so many ports available. The user does not have to enter the port number manually, cause it is selected randomly from numbers between 3000 and 3015. If a port is selected that is already in use, the process is automatically restarted until a free port is taken.

- ***Deployment and Integration:***

The microservice is going to be run- either by exporting it to .jar file, compilation and run through CLI or by importing (the latest version from git repository) into Eclipse IDE (2019-12) and pressing the run button.

It will be recognizable in the network by a combination of IP address, port number and endpoint. The message handler integrated in the mscfFramework will use this data to transfer the messages to microservice for processing.

- ***Testing and presentation:***

The Junit tests were carried out with the help of Junit 4.12. For this purpose, a separate test class was created, which is responsible for generating test data that is the same as the data that will come from the network. Therefore also data was generated which triggered exceptions. This tests helped finding mistakes and correcting those, which had as result the methods performing all of the functions that are expected.

Connection and sending basic messages tests will be done in a manual manner- at least two team members must be involved to try different scenarios and discuss whether the outcome is as desired or not.

- ***Status:***

The majority of the process is expected to be already ready for operation. After connecting the microservice to the network and receiving messages, it is expected that some methods will have to be revised or possibly rewritten. In the worst case, feedback could contribute that the process must be completely rewritten again, because the implementation does not match what was expected.

Implemented:

- CDGs give information on car positions and desired destinations,
- Simulate the traveling along a route provided by the ARS,
- Position updates are exchanged regularly, and off limit areas are taken in consideration,
- Multiple instances can work simultaneously,
- CDGs automatically and randomly select their start location and destination,
- Whenever a trip is completed the CDG instance select a new destination and repeat the whole automatic routing and traveling steps.

To be implemented:

- Passing and receiving objects from/to MessageHandler,
- Integration test,
- Behaviour in case an error occurred (e.g. not all fields in route drivable).

MS3 (Network Monitoring and Maintenance Service):

- **Design decision:**
  Our decision is to design the microservice with the help of the Spring Boot Maven application. We are going to use “Thymeleaf” dependency to present our collected information on the Website. We are using this open-source Java platform to easily build an web application. We want to create a web application linked to a relational database. All the philosophy we need to autoconfigure our project will be managed by the autoconfiguration library, which includes all the classes that are required to produce this type of application.

- **Deployment:**

  MS3 will communicate with the different microservices using MSCF jar/framework to forward messages to the wanted destinations through the endpoints that are defined in the messages, and also to store information of all transferred messages between the different microservices. The information in MS3 is based on a web visualization using the "mermaid" JS library. Running the spring boot application in Eclipse IDE will create a webpage on localhost 8080 with help of "Tomcat" that provides a web server environment in which data from Java can be displayed. The messages that are collected in the own SQL-lite database of MS3 will be written in Html file and display in the form of diagrams generated by the JS library.

- **Testing and presentation:**

  We are going to use the Junit5 framework as it is already described in some parts of the documentation, to test this microservice and all other microservices after we finish all the functionalities, of course, to be able to develop the nice running system. Definitely debugging this application through the methods of a standard Java exception is also an important part of testing our inner system.

- **Status:**

Implemented:

- Project Skeleton
- SequenceDiagram
- Thymeleaf function for HTML
- Simple Html example(HelloWorld Thymeleaf)

Not implemented:

- Unit testing
- SequenceController
- Database
- NMMController functionality
- MessageHandler functionality
- Functional Frontend with the interpretation of data

MS4 (Road Management Service):

- **Design decision:**

  MS4 is a java project that is created and managed with the use of Maven. One reason is that maven can automatically download the dependency libraries that are needed for the project. One such framework that we are using in ms4 is Spring boot. With the help of Spring Boot, a REST based web service is set up for the communication between ms4 and the CLI. IT provides a great deal of flexibility, reliability, and a hierarchical architecture between the components

- **Deployment:**

  The microservice is going to be run- either by exporting it to .jar file, compilation and run through CLI or by importing (the latest version from git repository) into Eclipse IDE (2019-12) and pressing the run button.
  After starting it, the application will start a webserver on the localhost under port 8084, over which the data between the CLI application and the ms4 can be exchanged.

- **Testing and presentation:**

  To test the basic function a simple framework will be used, which is the JUnit testing framework. Because the Unit tests can't test every functionality, by interacting with the other microservices, ms4 will be tested to see if it works as it is supposed to be. Testing and validating the REST service will be done using java rest-assured library. Further it will be tested with the use of Postman and the CLI application.

- **Status:**
  Majority of the functions are implemented but are not yet finished and will be changes.

Implemented
- RmsController: Endpoints and functions for processing commands and data from CLI

Not finished:
- CLI option search
- Blocking a part of a Street
- Unblocking of a Street
- Car Routes and Progress

Not implemented:
- JUnit and Integration tests
- Asynchronous Handling of messages