Distributed Systems Engineering
Submission Document
DEAD (Final Project)

Each team member should enter his/her personal data below:

Team member 1	
Name:	Moritz Renkin
Student ID:	11807211
E-mail address:	a11807211@unet.univie.ac.at

Team member 2	
Name:	Lukas Greiner
Student ID:	11807161
E-mail address:	a11807161@unet.univie.ac.at

Team member 3	
Name:	Isabel Pribyl
Student ID:	11807182
E-mail address:	a11807182@unet.univie.ac.at

Team member 4	
Name:	Nikita Glebov
Student ID:	01468381
E-mail address:	a01468381@unet.univie.ac.at

Birds View

Startup:

The Hub Operator must be started before all other MS because he checks if all Microservices are online. Only if all MS are online, the Hub Operator will give the start signal. (see “InstanceOnlineMessage” in the last chapter for more) These InstanceOnlineMessages will be sent over the messaging framework, like all other messages.

Next, the Hubs will dynamically exchange information about their neighbours using the HubConnectionInformation Message. This is necessary because each Hub only has information about its direct neighbours, as specified in the config file.

After this, the distributed system is ready to react to new containers at Source/Destination nodes.

Messaging Framework and Neighbour management:

Each Microservices must use the messaging framework (mf) for all communications. The only exceptions for this are the Rest APIs of the Source/Destination and Hub Operator which communicate with the Web interface.

In the context of the mf each message is sent out only to the direct “static”[1] and “dynamic” neighbours as defined in the configuration. The mf uses bridging to ensure that every microservice receives all messages. The containers dynamically change their neighbours depending on the current “owner” (e.g. a vehicle, hub). This makes sense, because in a real environment the distances between the nodes might be so far away that it becomes in efficient to bridge all messages to containers that have already been handed over to someone else.

When Source/Destination, Hub or Vehicle microservices hand over a container to another microservice, they deliver this information to the microservice as well as to the smart tag of the container. This way, even if a smart tag fails, due to travelling damages etc., the new “owner” of the container will still know its properties.

The Microservices use the Messaging Framework for setting up their communication. After initializing themselves the MF is ready to be integrated. The MS rely on the MF in their whole communication with each other using Callbacks for incoming messages and the Publisher to send messages.

[1] With static, we mean that the neighbours to not change; In contrast to dynamic neighbours (containers) which are handed over at runtime.

Lessons learned:

1. What were your experiences in the project?

We learned a lot, how microservices communicate with each other and mistakes, that can occur during development such systems as ours with usage of Multithreading. Also, it was exiting to take part in such big project. One of the challenges was to work as a team, where all pieces need to be coordinated, and also that experience is very valuable. We also understand the importance of tests, as on such projects often you can't easily simulate some behaviour to test new feature or to catch a bug.

2. What were the main challenges?

- Online Communication, no meetings in person
- Spring: The usage of spring for the first time was quite a challenge. As the dependency injection and the database creation/insertion is quite different from the old java way. But this challenge paid off, as the resulting code was much cleaner and easier to maintain.
- Network Delay and PC Performance: The bridging of the Messaging framework causes a big number of messages to be sent between the individual microservices at all times, even with our loop prevention working flawlessly. For the hubs, which have to bridge messages to all their vehicles etc., this caused issues because the messages came in faster than they could be sent out again for bridging, leading big delays in message transmission. Sending a message over a TCP connection some takes time (handshake, acks, etc.) especially with poor internet connection.
  To address this, we parallelized the socket creation and message-sending of the mf taking advantage of our multi-threaded environment.
  This solved the messaging issues, but created a performance overhead on our hardware, as we were hosting several microservices, each with a lot of worker threads. In the end we found the right balance between performance and networking capabilities by limiting the mf to 8 worker threads max, but it required a lot of time and effort for fine-tuning and testing.

3. Which challenges could be solved, and which could not?

All challenges could be solved, but network delay remains a problem if the internet connection of one team member is really bad (or crashes altogether).

4. Which decisions are you proud of?

- Callback based design in MF
- Using the Spring Framework to manage the Database and RESTful APIs

5. Which decisions do you regret?

- Dynamic Neighbour Handovers for bridging: As mentioned, containers will receive messages from different nodes along their way. We could have gone with the easy solution, keeping its connection to the Source Node it was initially generated. However, we decided not to do this because this would not be scalable in a real-world distributed system. The dynamic neighbour handovers have been implemented and they work fine most of the time (if all internet connections are fast) but our messaging framework was not designed with this in mind. Retrospectively, we should have included this in our planning from the beginning, or just leave it out altogether, for simplicity reasons.

6. If you could start from scratch, what would you do differently?

- The dynamic neighbour handovers, as described above in 6.
- Better time management, start early with development
    - Would have resulted in more time for end-to-end tests

Team contributions

Contribution of Member 1 (Moritz Renkin):

- Joint design of the overall architecture, messaging framework and microservices
- RestAPI design, together with Lukas Greiner
- Messaging Framework Implementation:
    - Message Delegation (Bridging, Loop Prevention)
    - Thread Logic
    - Network Service Initialisation and Termination
- Microservice Vehicle Implementation

Contribution of Member 2 (Lukas Greiner):

- Joint design of the overall architecture, messaging framework and microservices
- REST API design, together with Moritz Renkin
- REST API Implementation for the Hub Operator and the Station
- Hub Operator Implementation
- Station Implementation
- Web Apps for the Hub Operator and the Stations

Contribution of Member 3 (Isabel Pribyl):

- Joint design of the overall architecture, messaging framework and microservices
- Hub Implementation
- Container routing and optimisation logic implementation

Contribution of Member 4 (Nikita Glebov):

- Joint design of the overall architecture, messaging framework and microservices
- Messaging Framework: Neighbours Logic, Config Converter, few things in Network in and out with tests
- Microservice Container

Discussion - Messaging Framework:

- **Reflexion of the project and your progress (at most ½ A4 page):**

  Not many changes were made in compare to SUPD. New messages were added: ContainerPickUpRequest, DynamicNeighbourHandover, NewContainerAtSource, VehicleOffloadPermission.
  Logic of neighbours was slightly changed: neighbours were separated into Dynamic and Static to differentiate Microservices, that always communicate with the same neighbours and Microservices, that change their neighbours dynamically (Such as Vehicle). Handlers for Network In and Out were added to provide multithreaded connections.

  Sending and receiving messages works as expected, the check for duplicate messages performed by MessageUniqueenssChecker works as planned. The idea of using callbacks allowed us not only to simplify the implementation, but also showed excellent results in tests without causing any complaints.

- **Report on the project and implementation status (at most ½ A4 page):**

All functionality specified in the Assignment has been implemented. Messages are sent using Publisher and received using Subscriber with Callbacks. Every aspect of MF was tested with excellent results.

How To - Messaging Framework:

1. How to configure your service (file, format, processing in code etc.):

Each Microservice as: Hub, Vehicle, Source/Destination, Hub Operator has its own UUID. Microservice starts Massaging framework. Messaging framework uses argument such as:

1. Port
2. BridgingActivated Parameter
3. nodeType
4. Static neighbour's information:
    - UUID
    - Port
    - Ip
    - nodeType

It is using its UUID to convert data from config file, that is written in JSON format, stored on implementation folder.

```
"54b3a4a0-ef89-44f0-ac0c-f1f317bc3598": {  <———MS UUID
  "nodeType": "station",
  "bridgingActivated": true,
  "ip": "10.101.104.13",
  "port": 9009,
  "staticNeighbours": [
    {
      "uuid": "5f1822ca-8408-44b0-9ece-c397c73b1be2",
      "nodeType": "hub"
    }
  ],
  "comment": "Station ID: 8"
},
```

2. How to launch your service (tools, commands etc.):

Every Microservice starts Messaging framework with NetworkServiceFactory.intialize(UUID, Port (If it is Container Microservice)) method.

After that every Microservice should get its Publisher and Subscriber with getPublisher() and getSubscriber() method to be able to send or receive messages.

3. How to test your service (strategies, code, commands etc.):

Messaging framework was tested with Integrations and Unit Tests using JUnit.

The tests were carried out over:

- Config file converter
- Subscription to certain messages
- Message publish and subscribe Queues
- Check for message uniqueness
- Receiving and sending messages using Socket Programming

As well as testing the entire Messaging framework with Dynamic tests.

Interfaces & APIs - Messaging Framework:

1. **Reflexion of changes in comparison to SUPD (at most ½ A4 page):**

Messages added since SUPD:

- ContainerPickUpRequest
- DynamicNeighbourHandover
- NewContainerAtSource
- VehicleOffloadPermission

2. **Documentation of messages used internally by the MF:**
   Highlighted messages, that also used in MF:

- ContainerAtFinalDestination
- **ContainerHandover**
- ContainerPickUpRequest
- ContainerPositionUpdate
- **DynamicNeighbourHandover:** (This is the only network specific message, published and processed only internally in the mf)
- HubCapacityUpdate
- HubConnectionInformation
- InstanceOnlineMessage
- **NewContainerAtSource**
- VehicleArrival
- VehicleOffloadPermission
- VehicleOrder
- VehiclePositionUpdate

The detailed message descriptions are at the end of this document under “Documentation of the Messages & Communication of the MF”

Discussion - Microservice Containers [Smart Tags]:

- **Reflexion of the project and your progress (at most ½ A4 page):**

In compare to SUPD we removed listener for VehiclePositionUpdate. Due to the fact that this message was used to track the position of containers. However, this functionality was delegated to other microservices, so it was decided to remove this message.

- **Report on the project and implementation status (at most ½ A4 page):**

Container stores information about its UUID, weight, port, source location and destination location. Container updates ist current position as well as LocationHistory after becoming ContainerPositionUpdate. If container gets to it final destination, it self-destructs. Also, the container self-destructs after receiving the ContainerAtFinalDestination message.

How To - Microservice Containers [Smart Tags]:

- **How to configure your service (file, format, processing in code etc.):**

The ms requires following arguments to be passed as command line arguments:
1. UUID of container (UUID)
2. Weight (int)
3. Port (int)
4. UUID of current location (UUID)
5. UUID of destination (UUID)

- **How to launch your service (tools, commands etc.):**

Container microservice will be started by station microservice using its JAR file

- **How to test your service (strategies, code, commands etc.):**

There are several Junit tests, that covers functionality of container such as: current location change, Location history update. Container microservice stores current position every 3 seconds in separate file.

Interfaces & APIs - Microservice Containers [Smart Tags]:

1. **Reflexion of changes in comparison to SUPD (at most ½ A4 page):**

- VehiclePositionUpdate removed
- Added function, that writes in file named [UUID of container].txt position of container, in order to check the travel history of a container, in case it gets lost somewhere.

2. **Documentation of messages generated/sent by this MS:**

Sends/generated no messages.
Publishes:

- ContainerPositionUpdate
- ContainerAtFinalDestination

Discussion - Microservice Container Sources/Destinations:

We have renamed the Microservice “Source/Destination” to “Station” just to simplify the code and to avoid naming problems.

1. **Reflexion of the project and your progress:**
   The main difference to the SUPD plan is the addition of several separate message callback classes and additional classes for the REST API. Also several classes were removed as for example the “Station” and the “Location” classes as they had no additional value to the final implementation.

   The new callback classes are for the message type: ContainerHandover, ContainerPickupRequest and VehicleArrival. In the SUPD plan there was only the VehicleArrival Message Callback but after some time it emerged that they didn’t provide enough information to correctly execute the station functionality.

   The Location and Station class was removed as they were only encapsulating the same data as other classes and were therefore never really used.

   The REST API was split up to shorten the code of the original SUPD API class. This helped to improve the clarity of the API implementation in contrast to the original class.

2. **Report on the project and implementation status**
   The current status of the implementation of the Station looks like this:
   - The communication with the network is fully functional
   - The REST API is implemented but there are slight changes: The POST Mappings have been replaced with GET Mappings for easier implementation but the main functionality is fully provided
   - The web application is fully implemented and provides features for picking up and inserting new containers into the application
   - The web application shows all containers inserted into to the station and all containers to pick up from a station.
   - A new container process will be created by inserting a container via the web app

How To - Microservice Container Sources/Destinations:

1. How to configure your service (file, format, processing in code etc.):

To configure the service a config.json file must be provided in the same folder. It must include the following parameters:

- Object name: UUID of the microservice
- nodeType: must be station
- bridgingActivated: a station has to bridge messages – must be true
- ip: the ip of the service
- Port: the port the service is running on
- StaticNeighbours: the UUID and the type of all hubs directly attached
- Comment: is purely for easier human understanding and not used

2. How to launch your service (tools, commands etc.):

The service is simply started by this command:

- You have to be in the folder with the jar file and the config.json file
- java –jar ms-station-all.jar <UUID of Station>

The web application can be simply started with this command:

- You have to be in the project folder “station-view”
- You have to have npm installed
- Execute in the cli: “sudo npm install -g @angular/cli”
- Execute in the cli: “ng serve”

3. How to test your service (strategies, code, commands etc.):

When the application and the web application is started every station with their source and their destination repository can be accessed and managed in the web application. A new container can be inserted and if the container is at its final destination, it can be picked up there.

Interfaces & APIs - Microservice Container Sources/Destinations:

- **Reflexion of changes in comparison to SUPD (at most ½ A4 page):**

New Message Callbacks were added: CallbackContainerHandover, CallbackContainerPickupRequest and new listener classes were added to determine if a new container was inserted or a container arrived at its final destination.

The API was split up into the “Destination” and “Source” API where incoming requests are processed and added to the main repository. And the GET/POST Mappings were restructured so that they now have a standardised structure.

- **Documentation of messages generated/sent by this MS:**

Publisher of:
- InstanceOnlineMessage
- NewContainerAtSource
- ContainerAtFinalDestination
- VehicleOffloadPermission
- ContainerHandover

Subscriber of:
- InstanceOnlineMessage
- VehicleArrival
- ContainerPickUpRequest
- ContainerHandover

- **Documentation of the REST API for the WebUI:**

Compared to our SUPD design, we came up with more granular endpoints which lead to the web interface being able to get more specific info from the Source/Destination. For example, instead of having only one endpoints to get all containers currently stored in the Source Node, there is now the possibility, to get properties for a specific container.
An additional change to SUPD is that the UUID no longer has to be specified when inserting a container into the Source; This is no done be the Source/Destination node internally as requested in the SUPD feedback.

The detailed OpenAPI specification is also accessible in our Gitlab repository under “OpenAPI Specification/SourceDestination_APIReport.pdf”. The link below will open the visual representation of our specification.
GitLab - OpenAPI Specification

Discussion - Microservice Vehicles:

1. Reflexion of the project and your progress (at most ½ A4 page):

The vehicle uses a state-based design which, overall does not deviate a lot from the planning and skeleton submitted in SUPD. The states themselves, however, were adapted; Initially we thought that 3 States would be enough but during development we split the “ArrivalState” into “OffloadState” for loading off Containers and “PickUpState” for picking up containers at a Source. This separation makes sense with regard to the Single-Responsibility-Principle, as the two states have quite different requirements. Another advantage of a more granular state design is, that they are much better suited for unit testing.
For the driving simulation, we used the speed up factor 40, resulting in a driving time between Hubs of around 5-10 seconds. This seems to be a good middle ground between “waiting forever” and still being able to “see what is happening” in the Web interface.
Other than that, there were no major changes to the design of the vehicle ms.

2. Report on the project and implementation status (at most ½ A4 page):

All requirements of the Vehicle ms were implemented and judging by our unit tests, integration and end-to-end tests they are working perfectly.

Vehicles can receive orders from their respective Hubs, even save them for later if they are not yet ready for executing the order. Both orders to pickup and deliver containers are working flawlessly. Even in the first integration and end-to-end tests we found only minor bugs in this ms which was due to intensive unit-testing.

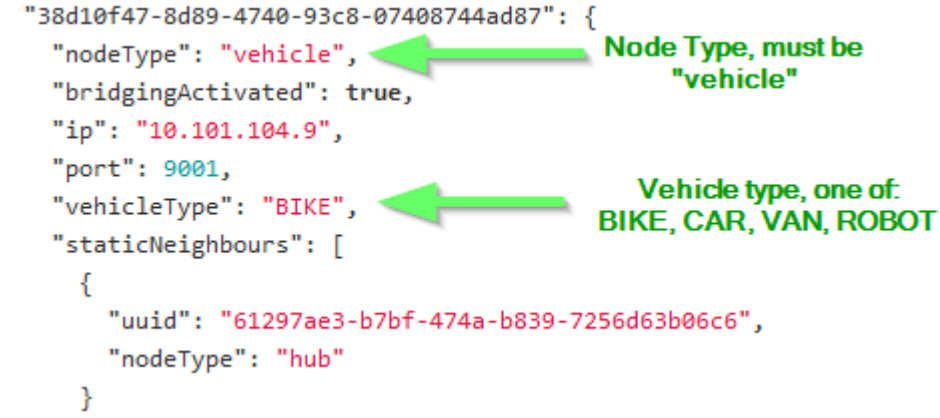
We conclude that the decision to use a state-based design was a good-one, as it allowed easy unit-testing and error detection.

How To - Microservice Vehicles:

1. How to configure your service (file, format, processing in code etc.):

Like the other microservices, the vehicle needs a config file in order for the messaging framework to work (see chapter How to - Messaging Framework).

Additionally, each vehicle needs to have “vehicleType” specified in the config. Here an example:



Speed and capacity of the vehicle are automatically derived from the vehicle type, as specified in the Assignment sheet and are therefore not configurable here.

2. How to launch your service (tools, commands etc.):

Like the hub and source/destination, the vehicle needs to be passed only the UUID as command line parameter. This UUID must be present in the config file.

To start all vehicles at once, we used a small python script (startAllVehicles.py) which reads all UUIDs in the config and starts each vehicle as a JAR with the respective UUID.

The vehicles must be started after the Hubs. (See “Birds View” for more)

3. How to test your service (strategies, code, commands etc.):

There are several unit tests and one comprehensive integration test available for this ms. All tests are written in JUnit 4, so they should be executable in any common IDE.

The integration test follows a scenario which covers most of the production code like initialisation, the right messages being sent out when arriving at a Hub, etc. It is a positive test, so it does not test the behaviour of the vehicle in case of bad input (non sensical messages, errors in the config etc.) This kind of negative tests are only done as unit-tests.

Interfaces & APIs - Microservice Vehicles:

1.Reflexion of changes in comparison to SUPD (at most ½ A4 page):

There were several changes in how vehicles publish/subscribe messages.
Most notably, we added the new message types ContainerPickupRequest, OffloadPermission, ContainerHandover.
Also, the Vehicle now sends out the ContainerPositionUpdate for the containers it transports.

2.Documentation of messages generated/sent by this MS:

Changes from SUPD are highlighted:

Publisher of:
- InstanceOnlineMessage
- VehiclePositionUpdate
- VehicleArrival
- **ContainerPickUpRequest**
- **ContainerPositionUpdate**
- **ContainerHandover**

Subscriber of:
- InstanceOnlineMessage (to wait for the Hub Operator to come online)
- VehicleOrder
- **OffloadPermission**
- **ContainerHandover**

The detailed message descriptions are at the end of this document under “Documentation of the Messages & Communication of the MF”.

Discussion - Microservice Hubs:

- **Reflexion of the project and your progress (at most ½ A4 page):**

  The Hub mostly was implemented according to the decisions made in the SUPD. In the SUPD the Callbacks in the diagram did not include a Callback for HubConnectionInformation message which turned out to be needed.

  In the SUPD, Routing was not really present in the diagram/skeleton. Therefore, a new package “routing” with a RoutingCalculator class was implemented for routing logic in the hub.

  To optimize vehicle filling, a vehicle only delivers orders or picks up containers if the vehicle can be filled with 50% if its capacity or a container is in storage for more than 5 seconds. This is done by adding a timestamp in the container which saves the time a container is added in the storage.
  A problem which rose while implementing was, how to prevent a hub from overflowing its storage and not have space for its own vehicles to off load, which would result in hubs blocking themselves. Since they cannot use their own vehicles to get containers out of storage. This was solved by reserving storage space of 40% for pickups and 60% for foreign vehicle arrivals. So, there is always space for own vehicles to off load.

- **Report on the project and implementation status (at most ½ A4 page):**

The hub fulfils all the functionality mentioned in the project description.

- **Organizing collection (from container sources):** to fulfil this, every hub calculates the shortest way to the destination and determines if it is responsible for pick up.
- **Delivering (to container destinations)**
- **Storing (when wating for transportation vehicle)**
- **Forwarding (between hubs):** if a new container arrives at a hub, the hub calculates its next hop.

To calculate the shortest route/determining next hop Dijkstra Algorithm is used.

How To - Microservice Hubs:

- **How to configure your service (file, format, processing in code etc.):**

Like the other microservices, the vehicle needs a config file in order for the messaging framework to work (see chapter How to - Messaging Framework).

In addition, the hub has the capacity given in the config file. The file also gets used to read out its own neighbouring connection to generate its map and exchange this information with the other hubs to generate the whole map.

```
{
  "61297ae3-b7bf-474a-b839-7256d63b06c6": {
    "nodeType": "hub",
    "bridgingActivated": true,
    "ip": "10.101.104.9",
    "port": 9001,
    "capacity": 5000,
    "staticNeighbours": [
      {
        "uuid": "5f4bcdd9-a857-4501-a479-8d511aba1dc5",
        "nodeType": "hub_operator"
      },
```

- **How to launch your service (tools, commands etc.):**

The MS needs to be started with its UUID as a start parameter. The UUID needs to be in the config.json in order to configure the hub with its information.

The Hub gets deployed with the help of a Python script startAllHubs.py. (See How To - Microservice Vehicles for more information).

- **How to test your service (strategies, code, commands etc.):**

The service mostly is tested with Unit Tests using JUnit 5. The Unit Tests cover Storage management functions, Routing (Dijkstra), Map Generation as well as the Main Controller functions. An Integration test with JUnit 5 was made to test how a container gets managed in the hub (from picking up from station, routing to forwarding to next station/hub). Mostly the info from the config file is used for easy set up.

Logging with sl4j also gives a great inside on what is happening inside the functions.

Interfaces & APIs - Microservice Hubs:

- **Reflexion of changes in comparison to SUPD (at most ½ A4 page):**

Very slight changes were made in comparison to SUPD. Instead of the message “StartSignal” the Hub waits for the InstanceOnlineMessage sent by the Hub Operator before starting its service.
The message HubCapacityUpdate served two purposes in the SUPD, sending an off-load permission to waiting vehicles and sending its storage capacity update for the Hub Operator monitoring service.
This message did not follow single responsibilty principle. The message was therefore split into two messages, HubCapacityUpdate (solely providing the storage capacity) and VehicleOffLoadPermission (solely needed by vehicles to determine if they can off load at the hub).

- **Documentation of messages generated/sent by this MS:**

Messages sent:

- InstanceOnlineMessage
- HubConnectionInformation
- VehicleOffLoadPermission
- HubCapacityUpdate
- VehicleOrder
- ContainerPositionUpdate
- ContainerHandover

The detailed message descriptions are at the end of this document under “Documentation of the Messages & Communication of the MF”.

Discussion - Microservice Huboperator:

- **Reflexion of the project and your progress (at most ½ A4 page):**

In the original SUPD plan a MySQL database was planned for the persistence part of the hub operator. After some research it was determined that an in-memory database (HSQLDB) is way better for our project as it was easier to use and to integrate. As spring in combination with hibernate has already standard support for HSQLDB the creation and the insertion into the database was much easier than anticipated.

Additionally, the hub operator got the functionality to wait for all other instances to send their InstanceOnlineMessage. After that itself sends a InstanceOnlineMessage to start and enable the whole container sending and receiving workflow.

Also, like in the Station Microservice, the REST API was split up into two. First the ContainerAPI and then the HubAPI. The mappings of the API endpoints was also refactored and are now in a standardized form.

- **Report on the project and implementation status (at most ½ A4 page):** The current status of the implementation of the Station looks like this:
    - The communication with the network is fully functional
    - The REST API Endpoints are all reachable via GET Requests and reply with the right data
    - The InstanceOnlineMessage is only sent when all other Microservices are online
    - The Database management is done fully automatically
    - The Insertion/reading into/from the database is fully functional
    - The Web Application shows all necessary information and is updating the data on reload

How To - Microservice Huboperator:

- **How to configure your service (file, format, processing in code etc.):**
  To configure the service a config.json file must be provided in the same folder. It must include the following parameters:

  - Object name: UUID of the microservice
  - nodeType: must be hub_operator
  - bridgingActivated: the hub Operator doesn't bridge messages – false
  - ip: the ip of the service
  - Port: the port the service is running on
  - StaticNeighbours: the UUID and the type of all hubs directly attached

- **How to launch your service (tools, commands etc.):**
  The service is simply started by this command:
  - You have to be in the folder with the jar file and the config.json file
  - java –jar ms-huboperator.jar

  The web application can be simply started with this command:
  - You have to be in the project folder "hub-operator-view"
  - You have to have npm installed
  - Execute in the cli: "sudo npm install -g @angular/cli"
  - Execute in the cli: "ng serve"

- **How to test your service (strategies, code, commands etc.):**
  When the application and the web application is started every Container with their source and their destination repository can be seen. The same workflow applies to the data of the hubs. There all their data can be seen and the current state can be loaded when the whole website is reloaded.

Interfaces & APIs - Microservice Huboperator:

1. **Reflexion of changes in comparison to SUPD (at most ½ A4 page):**
   There were some slight changes to the starting procedure of all microservices. Now the Hub Operator is responsible to send out a InstanceOnlineMessages when all other microservices have sent their InstanceOnlineMessages. This change made the correct starting procedure much simpler and more ellegant than anticipated.

   Also, the hub operator now uses the ContainerPositionUpdate instead of the VehiclePositionUpdate as this message contains all the necessary information to display the Container status in the web application.

2. **Documentation of messages generated/sent by this MS:**

   Publisher of:
   - InstanceOnlineMessage

   Subscriber of:
   - InstanceOnlineMessage
   - HubCapcityUpdate
   - VehicleOrder
   - ContainerPositionUpdate

3. **Documentation of the REST API for the WebUI:**

   The detailed OpenAPI specification is also accessible in our Gitlab repository under “OpenAPI Specification/ HubOperator_APIReport.pdf”. The link below will open the visual representation of our specification.
   GitLab - OpenAPI Specification

Documentation of the Messages & Communication of the MF

Note: All messages contain the UUID of the sending Microservice, even if not explicitly specified here. (Inherited from AbstractMessage)

Title	HubConnectionInformation
Description and Use Case	At the beginning of the setup the Hubs need to generate the whole map to route containers. Therefore, Hubs publish the Hubs/Stations they are connected to and receive this information from the other hubs.
Transport Protocol Details	Provided by the MF, message type: HubConnectionInformation

Created and Sent Payload	{
	"hub": "123e4567-e89b-12d3-a456-556642440000",
	"connections": "Collection<NodeConnection>"
	}
	
	Node Connection is provided by MF {"hub": boolean, "distance": int, "UUID": UUID}

Created and Sent Payload Description	• hub: A unique id of the hub, length of UUID, type: UUID, mandatory • connections: Lists the different connections of hub, which specifies the distance and if the connection is to a hub or station as well as their UUID, unlimited length. Type: Collection, mandatory
Relevant Response	-
Error Cases	-

Title	NewContainerAtSource
Description and Use Case	If a Station (Source) gets a new container in, it has to inform the hubs about the new container so they can pick it up. This message is also processed internally by the MF to ensure that messages are bridged to the new microservice.
Transport Protocol Details	Provided by the MF, message type: NewContainerAtSource

Created and Sent Payload	{
	"containerInformation":
	{
	"id": "123e4567-e89b-12d3-a456-556642440000",
	"weight": "5"
	"destination": "123e4567-e89b-12d3-a456-556642440000"
	"source": "123e4567-e89b-12d3-a456-556642440000"
	"in_hub": "boolean"
	"currHub": "123e4567-e89b-12d3-a456-556642440000"
	}
	"ip": "10.0.0.101"
	"port": 9001
	}

Created and Sent Payload Description	• ContainerInformation: contains information about the container (further explained in the following) •
Relevant Response	-
Error Cases	• Destination id doesn’t exist • Source id doesn’t exist

Title	HubCapacityUpdate
Description and Use Case	This message contains information about the current capacities of a hub.
Transport Protocol Details	Provided by the MF, message type: HubCapacityUpdate
Created and Sent Payload	{ "fillingLevel": 0.7 "maxCapcity": 5123 }
Created and Sent Payload Description	• fillingLevel: in percent type: double, mandatory • maxCapcity: the overall storage capacity, type: int, mandatory
Relevant Response	-
Error Cases	-

Title	VehiclePositionUpdate
Description and Use Case	The vehicle frequently sends out position updates, which the Hub Operator needs to monitor container locations and the container to update itself.
Transport Protocol Details	Provided by the MF, message type: VehiclePositionUpdate
Created and Sent Payload	{ "inTravel": "true" "distanceToGo": "0.2" "vehicleId": "some UUID" "originHub": "some UUID" "targetHub": "some UUID" }

Created and Sent Payload Description	InTravel: true if currently travelling somewhere, false otherwise. type: boolean, mandatory distanceToGo: distance which the vehicle needs to arrive at targetHub in km, type: double, mandatory vehicleId: type: UUID, mandatory OriginHub: The UUID of the hub where the vehicle started its journey from, type: UUID, mandatory TargetHub: The UUID of the hub which the vehicle travels to, type: UUID, mandatory Containers: A list of container Information (for further information look at message "New Container arrived at a station"), type: List<ContainerInformation>, mandatory
Relevant Response	-
Error Cases	Vehicle id doesn't exist

Title	VehicleOrder
Description and Use Case	If a hub sends out containers to another hub/end destination or picks up containers from a source. The Vehicle receives an order with the information what to do.
Transport Protocol Details	Provided by the MF, message type: VehicleOrder

Created and Sent Payload	{
	"targetHub": "123e4567-e89b-12d3-a456-556642440000"
	"containers": "List of container information" "distance": 12
	"pickup": false
	}

Created and Sent Payload Description	• targetHub: A unique id of the hub from where the vehicle starts the journey, length of UUID, type: UUID, mandatory • containers: A list of container Information (for further information look at message "New Container arrived at a station"), length of UUID, type: List<ContainerInformation>, mandatory • pickup: true if the vehicle should pickup containers at the destination, type: boolean • Distance: specifies the distance to the targetHub
Relevant Response	-
Error Cases	TargetHub id doesn't exist

Title	ContainerAtFinalDestination
Description and Use Case	The vehicle sends out that the container was delivered to its final destination.
Transport Protocol Details	Provided by the MF, message type: ContainerAtFinalDestination

Created and Sent Payload	{
	"containerInformation": "contains some container Information"
	}

Created and Sent Payload Description	• ContainerInformation: (for further information look at message “New Container arrived at a station”)
Relevant Response	-
Error Cases	-

Title	VehicleArrival
Description and Use Case	If the vehicle arrives at the hub/station is has to inform the hub that is here, along with the information about the containers that it carries. Is not sent when vehicles arrive at Stations for a pickup. Refer to ContainerPickUpRequest for more info.
Transport Protocol Details	Provided by the MF, message type: VehicleArrival

Created and Sent Payload	{
	"target": "123e4567-e89b-12d3-a456-556642440000"
	"origin": "123e4567-e89b-12d3-a456-556642440000"
	"containers": "List of some containers"
	}

Created and Sent Payload Description	1. Target: A unique id of the hub or station from where the vehicle has started its journey, length of UUID, type: UUID, mandatory 2. Origin: A unique id of the hub or station the vehicle has as its destination, length of UUID, type: UUID, mandatory 3. Containers: A list of container Information (for further information look at message “New Container arrived at a station”), length of UUID, type: List<ContainerInformation>, mandatory
Relevant Response	-
Error Cases	Target or origin hub id doesn’t exist

Title	ContainerPositionUpdate
Description and Use Case	Any Service currently transporting or storing a container send out this message to report the current location of the container to the Hub operator.
Transport Protocol Details	Provided by the MF, message type: ContainerPositionUpdates

Created and Sent Payload	{
	"containerInformation": "contains some container Information"
	"inTravel": true
	}

Created and Sent Payload Description	1. ContainerInformation: (for further information look at message “New Container arrived at a station”) 2. inTravel: true if the Container is currently travelling in a vehicle, type: boolean
Relevant Response	-
Error Cases	Container doesn’t exist

Title	InstanceOnlineMessage
Description and Use Case	This message has to be sent by all microservices when they boot up. However, they shall not start operation until the Hub Operator sends its InstanceOnlineMessage. The Hub Operator will not send its InstanceOnlineMessage until it received it from all ms.
Transport Protocol Details	Provided by the MF, message type: InstanceOnlineMessage

Created and Sent Payload	{
	"type": "type of MS"
	}

Created and Sent Payload Description	• type: The type of the MS which sends this message out, type: Enum of the MS type, mandatory
Relevant Response	-
Error Cases	The hub operator does not recognize/receive all InstanceOnlineMessages and never sends the start signal.

Title	ContainerHandover
Description and Use Case	This message must be sent out by MS handing over a Container to another MS. Even if this message is not subscribed by any other service, it is processed by the Messaging Framework.
Transport Protocol Details	Provided by the MF, message type: ContainerHandover

Created and Sent Payload	{
	
	"takerId": " UUID"
	
	"containerId": "UUID"
	
	}

Created and Sent Payload Description	• takerId: UUID of the Microservice that will now take the container in its "possession", type: UUID, mandatory • containerId: UUID of the container, type: UUID, mandatory
Relevant Response	-
Error Cases	

Title	ContainerPickUpRequest
Description and Use Case	Sent out by vehicle to signal arrival at a source node to pick up certain containers.
Transport Protocol Details	Provided by the MF, message type: ContaienrPickUpRequest

Created and Sent Payload	{
	
	"stationId": " UUID"
	
	"containersToHandover": Collection<containerInformation>
	
	}

Created and Sent Payload Description	• stationId: UUID of the station that should handover the containers, type: UUID, mandatory • containersToHandover: the specific containers to be handed over, type: UUID, mandatory
Relevant Response	-
Error Cases	The source does not have the containers anymore.

Title	vehicleOffloadPermission
Description and Use Case	Published by Destinations and Hubs after vehicles arrived to signal that there is enough free capacity for the vehicle to handover its containers.
Transport Protocol Details	Provided by the MF, message type: ContaienrPickUpRequest

Created and Sent Payload	{
	"vehicleId": " UUID"
	}

Created and Sent Payload Description	• vehicleId: UUID of vehicle to hand over its containers, type: UUID, mandatory
Relevant Response	-
Error Cases	

Title	DynamicNeighbourHandover
Description and Use Case	Only for internal use for Messaging Framework. The messaging framework sends out this message if the microservice publishes a container handover. It looks up the IP and port for that neighbour and removes it from the neighbours. The receiver of the message will add the handed over neighbour with the specified ip and port.
Transport Protocol Details	Provided by the MF, message type: ContaienrPickUpRequest

Created and Sent Payload	{
	"takerId": " UUID"
	"neighbourId": " UUID"
	"neighbourIp": 10.0.0.101
	"neighbourPort": 9001
	}

Created and Sent Payload Description	• UUID of the Microservice that will now take the neighbour for bridging, type: UUID, mandatory • neighbourId: the UUID of the handed over neighbour • neighbourIp: the IP of the neighbour, type: InetAdress • neighbourPort: port of the neighbour, type: int
Relevant Response	-
Error Cases	The messaging framework cannot find ip and port of the neighbour