DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

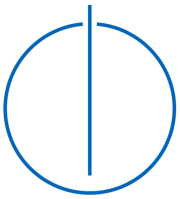
Chair of Computer Architecture and Parallel Systems

BMW Map Service

Project Documentation
IoT Lab Course Winter Term 2018/2019

Aljoscha Kullmann, Okan Kamil Şen, Dominik Serve, Cédric Stark

February 17th 2019



Contents

1 Project Scope **4**

2 System Overview **5**

3 Cloud **6**
3.1 Components . . . 6
3.1.1 API . . . 6
3.1.2 Visualization . . . 7
3.1.3 Standard Components . . . 7
3.2 Deployment . . . 7
3.2.1 Installation . . . 7
3.2.2 Start Up . . . 8
3.2.3 Initialization . . . 8
3.3 Usage . . . 8

4 Visualization **9**
4.1 Architecture . . . 9
4.1.1 Map and Timeline Slider . . . 10
4.1.2 MQTT Live Updates . . . 10
4.1.3 Replay Buffer . . . 10
4.2 Deployment & Usage . . . 10
4.3 Gamification . . . 10

5 Robot **12**
5.1 Common Components . . . 12
5.1.1 ROS Bridge . . . 12
5.1.2 Slicer and Stitcher . . . 12
5.2 Map Updates . . . 13
5.3 Simulators . . . 14
5.3.1 Automatic Simulator . . . 14
5.3.2 Interactive Simulator . . . 14
5.3.3 Loadtesting . . . 14
5.4 Deployment . . . 14

6 Evaluation **16**
6.1 Resource Requirements . . . 16
6.2 Known Limitations . . . 16

7 Ideas for Future Work **18**

8 Conclusion **19**

A Comprehensive API Documentation **20**
A.1 Chunk Service . . . 20
A.1.1 Meta Data . . . 20
A.1.2 Chunk REST Interface . . . 20
A.1.3 Chunk MQTT Interface . . . 23
A.2 Coordinate Service . . . 24
A.3 API Configuration . . . 25
A.4 API Development . . . 26

A.4.1 npm link 26
A.4.2 Linting 26
A.4.3 Testing the chunk service 26

B Visualization Setup for Development **28**
B.1 Working with Vue.js 28
B.2 Configuration 28
B.3 Configuring the IDE 28
B.3.1 Installation 28
B.3.2 IDE Settings 29
B.3.3 Breakpoints in VSCode 29
B.3.4 Vue.js Guides 29

Abstract

This is a comprehensive documentation for the BMW map service project of the IoT practical course in winter semester 2018/2019. Additionally to the usage and API instructions present in the GitLab repository, this document features a detailed description of the architecture, as well as the evaluation of the developed solution and possible future improvements.

1 Project Scope

BMW requested a map service solution that enables autonomous robots to share their knowledge about their environment, the factory work floor. The service should also come with a visualization board that allows maintenance workers to see the current map with the robots on it and also replay the information for up to a week.

Initially, there was a solution from summer semester 2018 to build upon. The former project implemented a cloud service that enabled robots to synchronize and update their map, as well as the display of a live stream of the current map state and robot positions.

The improved map service should most importantly save a history of map updates and information about the robot, for a time span of seven days. Furthermore, it is supposed to be able to chronologically replay the map and robot updates. All that is done using a video-like visualization, allowing to reproduce the map updates at different playback rates and including a live view.

Since the map synchronization between robot and cloud was already realized by the last team and BMW already possesses similar services, the focus was less on the robot part. The main effort was to update the cloud and create an appropriate visualization in order to comply with the requests by BMW. The requested features were:

F.1 Save history of map updates

F.2 Save history of system data (robot position, status, battery level, planned path)

F.3 Ability to replay map updates and system data in chronological order

F.4 Play (Single / Double / 4x / 8x speed)

F.5 Pause

F.6 Select starting time (hours – minutes – seconds) within last 7 days

F.7 20 robots send update every second

Not only were these features implemented successfully, but also different possibilities were elaborated to simulate the robot output in order to conduct performance tests and a gamification approach was developed in order to present the idea in a more appealing fashion.

2 System Overview

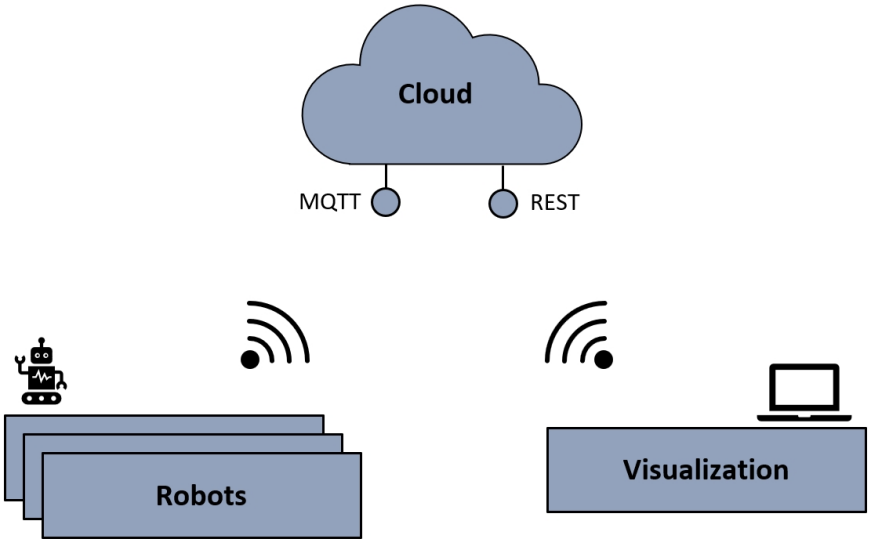


Figure 1: Core components of the newly developed solution.

The map service can be split into three core components: the production floor robots, the client-side visualization, and the cloud connecting the whole system. The cloud provides an API that allows to publish and subscribe map and robot updates, as well as to perform sophisticated queries. Robots send and receive map updates and publish their status information. The visualization component only reads data from the cloud.

3 Cloud

3.1 Components

The cloud is composed of different services that can be deployed as docker containers. The customized components are the *coordinate-service* and the *chunk-service*. The *visualization* is served by the cloud, as well. Finally, the cloud makes use of some standard components: an *mqtt broker* and a database (elasticsearch). nginx serves the built visualization and the custom API services.

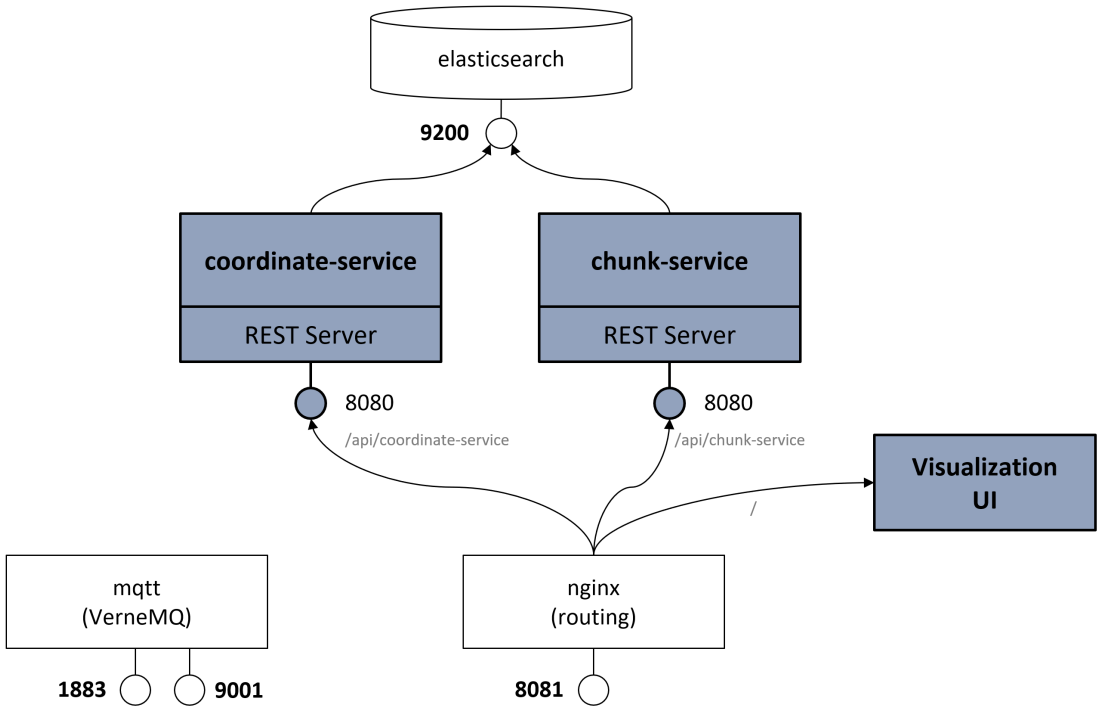


Figure 2: Architecture of the cloud service, including the visualization component. Blue boxes indicate customized components, whereas white boxes are standard software. All boxes are deployed as docker containers. Bold port numbers are visible outside of docker.

3.1.1 API

The custom API offers two access points: via MQTT (port 1883) and via HTTP (REST, port 8081). MQTT is used for asynchronous communication, namely chunk and robot information updates. REST Requests come either from robots trying to download the map on start-up or the visualization component.

You can find a detailed documentation of the API in appendix A.

Map updates are processed by the *chunk-service*, which validates the chunk data, advertises the update to the other robots, and stores the map update in the Elasticsearch database. Map chunks are uniquely identified by their `row`, `column` and `timestamp` attributes. They are stored both as Strings that are B64 encoded png and that are pgm B64 encoded and compressed.

The robot information updates are handled by the *coordinate-service*, which stores them in Elasticsearch directly.

3.1.2 Visualization

The visualization is scaffolded with Vue CLI 3[1]. A multi-stage docker container is used to build and deploy the visualization. The build-stage installs the required npm packages and runs the `npm run build` script to minify the visualization to a static package. The production-stage uses an nginx docker image to serve the app from the `/usr/share/nginx/html/` directory. The application is client-side only, so no session is kept, but the client will autonomously query the services and MQTT server for updates.

3.1.3 Standard Components

The cloud includes an Elasticsearch database docker image in version 6.5.4 that is accessed only through the service components on port 9200. The MQTT broker is VerneMQ version 1.6.2, available to all system components on port 1883.

3.2 Deployment

The Cloud also serves the visualization. So, this section also covers the deployment of the visualization.

3.2.1 Installation

1. We recommend to use a machine running Ubuntu. (We used Ubuntu 14.04 as well as 18.04.)
2. Set up Docker CE for Ubuntu as described here. (We used Docker v18.09.)
3. Recommended: Install node.js from the NodeSource repository as described here.
4. Recommended: Open the required ports if you want to access the cloud from outside your machine. The commands can be found in listing 1.

```
$   sudo ufw enable
$   sudo ufw allow 8081
$   sudo ufw allow 1883
$   sudo ufw allow 9001
$   sudo ufw allow 9200
```

Listing 1: Commands to open the required ports.

5. Clone this repository

```
$   git clone https://gitlab.lrz.de/iot-bmw-map-service-18w/cloud.git
```

Listing 2: Address to clone the repository from.

6. Recommended: add a file named `config.json` to the `./visualization` directory if you want to access the visualization from outside your machine. This file should look like this:

```
{
    "API_URL": "http://<IP-OF-YOUR-MACHINE>:8081",
    "MQTT_URL": "<IP-OF-YOUR-MACHINE>:9001/mqtt"
}
```

Listing 3: Content of the config.json for the visualization.

[1] `https://cli.vuejs.org/`

7. Run `docker network create custom_network`

Note: The visualization uses a socket version of mqtt. This is why it uses port 9001.

3.2.2 Start Up

To start up the cloud, run the following commands in the repository folder:

1. `sysctl -w vm.max_map_count=262144` (this is needed by elasticsearch)
2. `docker-compose up --build`

For convenience, you can also just run `./start.sh`.

3.2.3 Initialization

Although not always necessary, it is recommended to upload a map, initially. Therefore, conduct the following steps:

1. Download the `big_map.zip` from here.
2. Extract this file to `./api/chunk-service/data`.
3. Open a shell and execute the following commands.

```
$   cd api/chunk-service
$   npm install
$   npm run fillDB
```

Listing 4: Commands to upload an initial map.

3.3 Usage

- Navigate with a browser to `<IP-OF-YOUR-MACHINE>:8081` to view the visualization board.
- Navigate with a browser to `<IP-OF-YOUR-MACHINE>:8081/controller` to control your own robot. Note: this requires a map to be uploaded to the cloud as described in the section 3.2.3.
- Interact with the API as described in appendix A.
- To delete the database, run the following commands:

```
$   curl -X "DELETE" <IP-OF-YOUR-MACHINE>:9200/meta
$   curl -X "DELETE" <IP-OF-YOUR-MACHINE>:9200/chunks-latest
$   curl -X "DELETE" <IP-OF-YOUR-MACHINE>:9200/chunks
$   curl -X "DELETE" <IP-OF-YOUR-MACHINE>:9200/coordinates-latest
$   curl -X "DELETE" <IP-OF-YOUR-MACHINE>:9200/coordinates
```

Listing 5: How to delete the database.

- You can also configure an automatic data rollup. This for example deletes all data that is older than a week. For more information have a look here.

Note: If you are accessing the cloud from your local machine, `<IP-OF-YOUR-MACHINE>` will of course be `localhost`.

4 Visualization

4.1 Architecture

Figure 3: Screenshot of the visualization board.

The visualization is a Vue.js[2] web app on Node.js. Live data about the map and the robot positions is collected via MQTT, historic data is queried from the cloud through REST queries. The core of the visualization component is a buffer (shown in Fig. 4 that contains the map updates to be displayed in the coming ten seconds. Depending on the replay speed, upcoming map changes are buffered and then applied on the map.

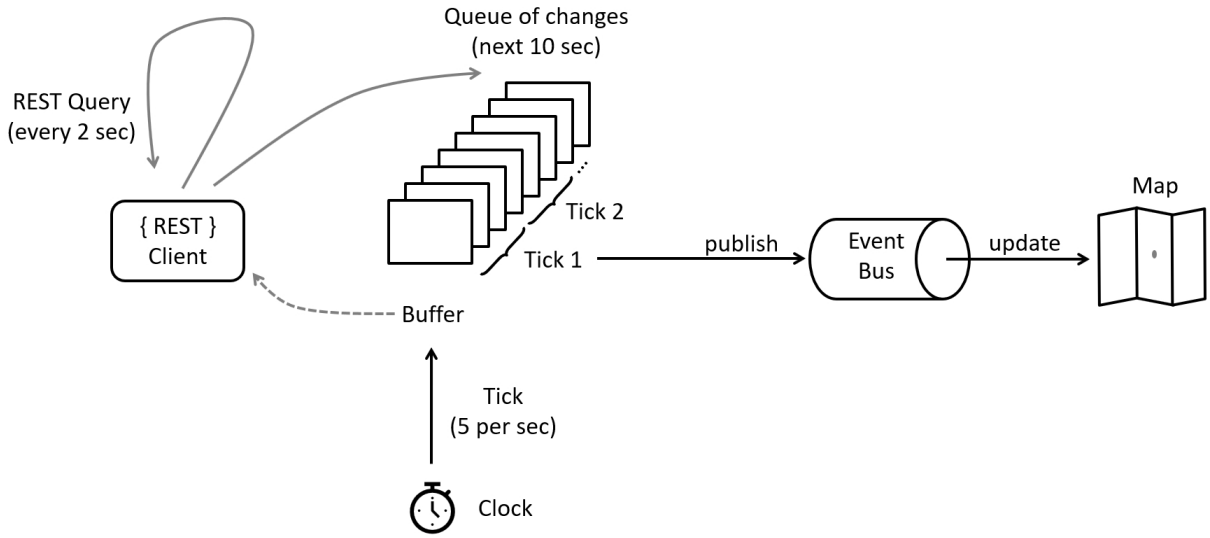


Figure 4: Core feature of the visualization component is a buffer.

[2] https://vuejs.org/

4.1.1 Map and Timeline Slider

The map view of the visualization shows a zoomable canvas of map chunks, overlaid with the robot SVG vector graphic representations. On the bottom of the page, the user can choose the current timestamp on a timeline slider, up to the current time. The map chunks and robot information are queried for the current timestamp. If the timestamp lies in the past, the chunk and robot information for the next 10 seconds is also buffered. If the timestamp is the current time, *live mode* is activated and updates are read directly from their MQTT topics. The time-span can be chosen from several steps, e.g. 1 hour or 1 week back. The visualization can be replayed and paused, as well as cycled through different speed levels, e.g. double, 4x, $1 min/s$ (60x) or $1 day/s$ (86400x). The visualization updates at a rate of 5 frames per second. The current timestamp and the duration to the current real-time is also displayed on the slider.

4.1.2 MQTT Live Updates

The *Live mode* is the easiest because after loading the latest data, the client only has to forward the advertised updates from mqtt to the internal *Event Bus* which is also described in the next section. The replay functionality comes with a completely different set of challenges. This is why the buffer is of focus in the following.

4.1.3 Replay Buffer

Fig. 4 shows the Replay Buffer. The global clock is updated five times per second (configurable frame rate). It is published via the Vuex[3] Store, as are the map metadata and the complete map after a complete reload of the map at certain timestamps. The buffer listens to these ticks and initiates REST queries every two seconds to fill a queue with the map changes for the next ten seconds. Finally, with every tick the respective changes from the queue will be published via an Event Bus that updates the map.

4.2 Deployment & Usage

The visualization is served by the cloud. Refer to sections 3.2 and 3.3 for more information.

4.3 Gamification

A simple interface to simulate a robot is also accessible through your browser. Navigate to `<IP-OF-YOUR-DEPLOYED-MACHINE>:8081/controller` to open a javascript simulator. You can move the robot using the *UP*, *DOWN*, *LEFT*, *RIGHT* buttons and send chunk updates using the *BLACK*, *GRAY* and *WHITE* button.

Fig. 5 shows a screenshot of the simulator interface.

[3] `https://vuex.vuejs.org/`

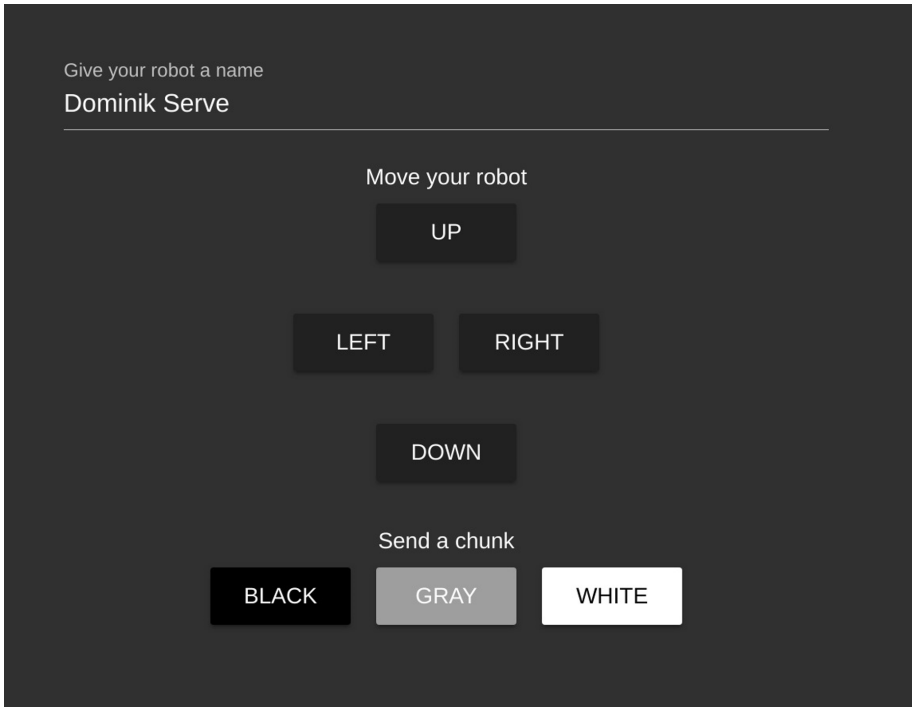


Figure 5: Screenshot of the user simulated robot.

5 Robot

5.1 Common Components

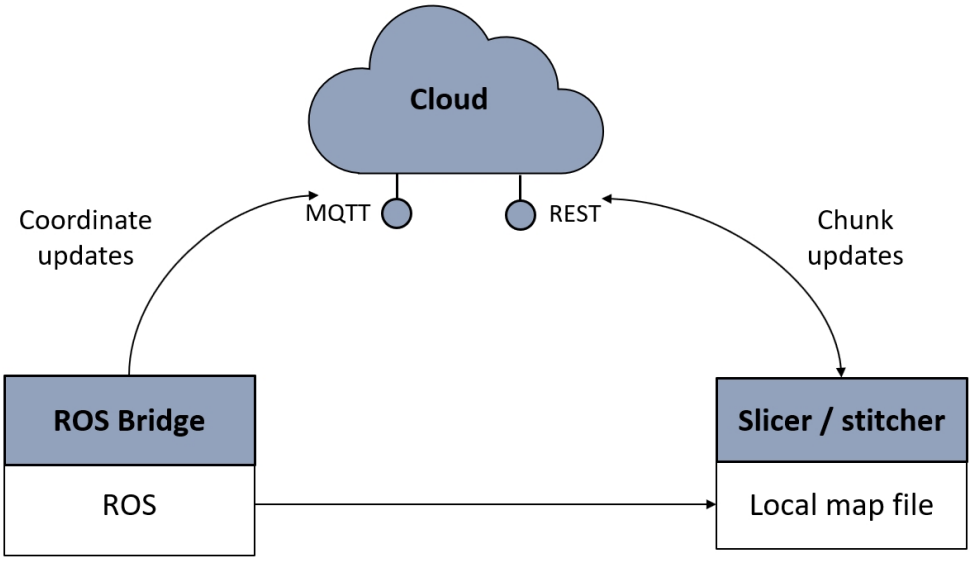


Figure 6: Architecture of the robot. ROS and the ROS Bridge can be replaced by a suitable simulator. Components in grey boxes have been newly created, while white boxes indicate standard components.

As shown in Fig. 6, the robot can be split in two component groups, which are responsible for different tasks.

5.1.1 ROS Bridge

ROS sends robot status information via odometry messages using the MQTT protocol. The `ROS Bridge` is subscribed to the corresponding topic and receives those messages. After some filtering and the addition of a timestamp, these messages are sent to the cloud, again via MQTT. The status updates are sent once every second, which is industry standard. The map updates are handled the same way in the simulator and with ROS, see Sec. 5.2.

For testing and demonstration purposes, the ROS and ROS Bridge part can be replaced by a simulator that creates map and robot information updates, refer to Sec. 5.3 for more details.

5.1.2 Slicer and Stitcher

`Slicer` and `stitcher` are the central piece of the map update functionality. As the basic idea behind this project is to only update *chunks*, and not the complete map, we need a possibility to split the map into smaller parts, as well as putting it back together. `Slicer` and `stitcher` perform exactly that. Additionally, they take care of compressing/decompressing the chunks. Both programs are used by `main.py`, which takes care of comparing the chunks and deciding which updates to send. To manually create slices, use the following command:

```
$  python slicer.py \
$  --file map.pgm \
$  --output-dir /home/user/output \
$  --chunk-width 100 \
$  --chunk-height 80
```

Listing 6: Command to manually slice a map.

The `slicer` is taking the file specified with `file`, slices it into chunks of the dimensions `chunk-width` and `chunk-height` and saves them to `output-dir`. Chunk width and height have to divide the `PGM` width and height without rest.

To recreate a complete map, run the following command:

```
$  python stitcher.py --working-dir /home/user/output
```

Listing 7: Command to manually stitch chunks back together.

Here, `working-dir` is supposed to contain the chunks, as well as a file called `meta.json`, which has this structure:

```
{
   "comment":"# CREATOR: GIMP PNM Filter Version 1.1",
   "width":4800,
   "height":3200,
   "max_brightness":"255",
   "CHUNK_WIDTH":100,
   "CHUNK_HEIGHT":100,
   "CHUNK_ROWS":32,
   "CHUNK_COLS":48
}
```

Listing 8: Content of `meta.json`.

5.2 Map Updates

The map synchronization is handled using the `slicer` and `stitcher`. When starting the robot for the first time, it queries the cloud for an existing map. If one is found, it receives a REST response containing the map, if not, the robot uploads its local map. To keep the robot map up-to-date, the following steps are performed:

- **Integrate external chunks:** Chunk updates by other robots are advertised by the cloud, the robot integrates them into its local map. The other two steps are skipped in this iteration.
- **Publish internal updates:**
  1. **Check for local changes:** The robot periodically compares the current state of the local map with the one from the previous iteration. If a map chunk has changed more than a specified threshold, it is appended to an additional list of chunks to be published.
  2. **Publish local updates**: The chunks in the aforementioned list are published via MQTT.

In order to be able to access the ROS map, it has to be exported from the navigation stack. The resulting `PGM` file can then be synchronized with the cloud. For ROS to use the updated map, the navigation has

to be shut down and restarted. Since shutting down the robot should not happen arbitrarily, but only when the robot is safely parked, the decision was taken not to automate this procedure in a script, but to let BMW handle this part.

5.3 Simulators

While ROS offers many different simulators, we decided to implement several, more lightweight, options for this project.

5.3.1 Automatic Simulator

The automatic simulator simulates map changes as well as robot movement. The robots are moving in circles, each with an own, random center point. This simulator can be run in Docker or manually (see Sec. 5.4) and provides the best results to test the visualization, since it automatically creates data and changes to the map can be seen. It also allows you to simulate multiple robots at the same time.

5.3.2 Interactive Simulator

The interactive simulator allows you to control your own robot. Start it using

```
$  python3 src/interactive_sim.py
```

Map updates can be activated, too. In order to send them to the cloud, additionally run the chunk-manager manually, without the `simulator` flag (refer to Sec.5.4). This simulator has the advantage, that you see the changes happening to the map. The same environment variables are used as for the automatic simulator, for details refer to Sec. 5.4.

5.3.3 Loadtesting

For loadtesting, a simple script, `simple_simulator.py` was developed. The test results can be found in detail in Sec. 6.1. This script has to be modified in order to set the number of robots and address of the cloud, as it is not using environment variables. Furthermore, while chunk updates are sent, these are just the same old chunks, except that they are assigned to a new position. For that to work, a directory containing map chunks has to be provided and specified in the script. This simulator is the most lightweight option, and thus easily capable of simulating considerably more robots at the same time than the first two simulators.

5.4 Deployment

The robot can be run manually by executing `start_without_docker.sh`, or (in combination with the automatic simulator) using docker-compose with the following command:

```
$  docker-compose up --build --scale robot=<NUMBER OF ROBOTS>
```

In both cases, you can modify `start_without_docker.sh` and `docker_entrypoint.sh` to set the following flags:

- `file`: map file
- `meta`: meta file
- `output-dir`: output directory for sliced chunks
- `chunk-height`: chunk height in pixels
- `chunk-width`: chunk width in pixels
- `debug`: set to `true` to show debug logging
- `simulator`: set to `true` to simulate map updates and robot movement

Running the robot in Docker has the advantage that the original map file remains untouched, however, changes in the robot map can obviously not be seen, as you don't have direct access to the map file. To see the map from the robot perspective, you have to run it in manual mode, which will modify the original file.

The following environment variables have to be set for the robot to work correctly. These can be either set in `docker-compose.yml`, or exported as shell environment variables, when the robot is run manually.

- `API_HOST`: hostname of the cloud server
- `API_PORT`: port of the API, usually 8080
- `MQTT_BROKER`: hostname of the MQTT broker, included in the cloud, so this is usually the same address as `API_HOST`
- `MQTT_PORT`: port of the MQTT broker, usually 1883
- `CHUNK_WIDTH`: width of a single map chunk, e.g. 100
- `CHUNK_HEIGHT`: height of a single map chunk, e.g. 100
- `UPDATE_FREQUENCY`: number of coordinate updates per second, usually 1

6 Evaluation

6.1 Resource Requirements

In order to determine the resources needed to run the cloud in a realistic environment, the `simple_simulator.py` script was run for one full day. 20 robots sent a coordinate update each second, every 5 seconds each robot is additionally sending a chunk update.

For the test, three different machines (one for the cloud services, one for elasticsearch, one for the robots) were used. That way, the influence of elasticsearch could be measured separately, a component which, presumably, would be responsible for most of the resource usage. All machines were *Google Compute Engines*, both the Iot cloud and elasticsearch running on VMs with 2 vCPUs and 10.5 GB of memory each, the OS was Ubuntu 14.04.

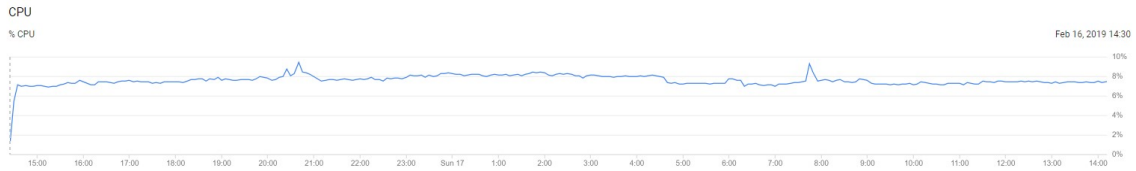


Figure 7: CPU usage on the IoT cloud machine during the experiment.

Figure 8: CPU usage on the elasticsearch machine during the experiment.

As one can see in Fig. 7, the load created by the communication, the coordinate- and the chunk- service is relatively small, remaining at roughly 8% CPU usage on the IoT machine. As expected, elasticsearch was the more resource intensive component (see Fig. 8), with an average CPU usage of 60% during the experiment.

The storage usage of elasticsearch is shown in Fig. 9, overall a consumption of 300 MB per day can be observed. The quite significant decrease in storage usage after roughly 14 hours is most likely due to elasticsearch compressing the database. Most notably, the space saved is entirely freed up by the coordinate index (brown line). This makes sense, as the chunk data (green line) is already transmitted in a compressed form, and can thus not be further reduced in size by elasticsearch

To initially fill the map (Sec. 3.2.3), more than 100 HTTP connections were created simultaneously, without having caused any issue, proving that even more than the 20 robots used in this experiment can be connected to the cloud.

6.2 Known Limitations

The cloud can be scaled in different ways, most easily by spreading the different microservices to different machines, as done in the experiment in above. Furthermore, it could be deployed in Kubernetes, which, however, is not done in the current implementation. The experiment showed that elasticsearch is the

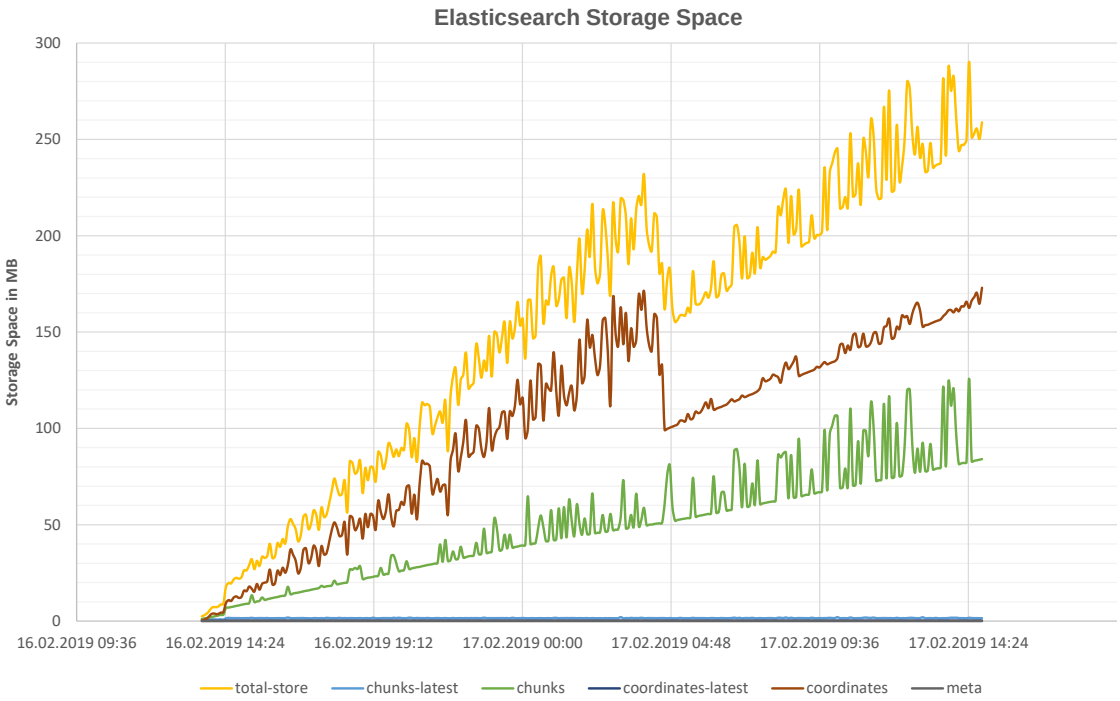


Figure 9: Record of the storage used by elasticsearch during the experiment. The yellow line shows the total space and the other lines refer to the different indices.

most resource intensive component by far, deploying it in Kubernetes is described here and should take care of most of the potential resource shortages one might experience.

The Visualization supports all the functionality that was defined in sec. 1. However, the playback functionality can sometimes skip some frames. Also, the backwards functionality of the visualization still has some bugs. The replay feature is nice. But we found that it sometimes can be hard to find chunk changes if they don't happen too often. This is why we support faster replay speeds and suggest some more features as described in I.1 of sec. 7.

On the robot side there are also a few limitation to take into account. `Slicer` and `stitcher` only support absolute paths, relative paths might lead to undefined behaviour. The simulators use the *Netpbm* toolkit to simulate changes in the map.

The integration with ROS currently only includes the odometry messages, to update the map the whole navigation stack has to be shut down. The map can then be synchronized with the cloud, and the navigation restarted with the new data. A way to update the map while ROS is running the navigation was not found, this is also the way BMW is currently using the system.

7 Ideas for Future Work

The following list collects some ideas that came up during this project on how to continue the work.

I.1 **User Interface.** There were some more features envisioned that could not be finished in the scope of this project:

- Indicate the ocurrences of chunk changes in the slider as suggested by the Mockup.[4]
- Let users show and hide different kinds of information in the visualization, such as the names, battery status and path of each robot or the borders, update timestamps and robotId of each chunk.
- Make the time range of the slider more flexible. Allow the user to "zoom in" at any point in time.
- An interesting addition to the visualization would be the possibility to manipulate the map. That way, users could create POIs, or declare certain forbidden zones.

I.2 **Scalability.** Make the cloud scalable by deploying the cloud compontents in a kubernetes cluster. The containers published in the container registry might be of interest for that.

I.3 **Time Synchronization.** It was a design decision to use the timestamps sent by the robots within the cloud. This is because of the asynchronous nature of MQTT. Robots should have their time synchronized on system level. However, it would be interesting to investigate the feasibility of that aproach.

I.4 **Detect Conflicts.** Currently, the cloud accepts all chunk updates as long as they have a newer timestamp than the latest one and the pgm B64 compressed string is not malformed. It would be interesting to implement a mechanism that could detect conflicts between two robots overwriting their previous updates.

I.5 **Chunk Comparison.** Slicing, stitching and chunk comparisons require relatively much CPU power. To allow for faster comparisons of map chunks, one could hash the chunks, possibly in a Merkle tree structure for binary search of changes[5].

I.6 **Path Planning.** Robots could publish their path and users could be allowed by the visualization to interfere.

I.7 **Explore different Technology.**

- Elasticsearch does not naturally support queries like *get latest chunks updated before....* This means that for big databases the performance of the visualization queries might go down. Databases like Graphite[6] might be an interesting alternative.
- Also, elasticsearch does not support to store binary data. It always has to be base64 encoded. Databases like CouchDB[7] might be an interesting alternative to store the png data.
- Although not of use for the current robots at BMW, we find dockerized robots interesting: the IoT toolset Balena[8]. This would mean: ROS, mission manager and network communication would run in separate container images and could be exchanged and updated on-the-fly.

[4] *Note:* It would not be wise to download all chunk changes at once and to store them in the visualization permanently. But one could use pagination and download the `reduced` data from the chunk service (see here). You could draw lines in an html canvas indicating the chunk changes. Do this for every REST response for pagination and discard the data afterwards.

[5] `https://en.wikipedia.org/wiki/Merkle_tree`

[6] `https://graphiteapp.org/`

[7] http://couchdb.apache.org/

[8] `https://www.balena.io/`

8 Conclusion

It is not always an easy task to build upon an existing project. The first challenge is to get the old systems running. Secondly, you have to dive into the code and understand the architecture. After making a plan, some components may have to be redone because the old architecture does not support certain features and becomes obsolete. Only then can you start creating new features. This is why own expectations on how many new features can be achieved may have to be lowered. Nonetheless, we are proud that we accomplished all the features (F.1 – F.7) that were defined in the project scope (sec. 1).

One of the objectives of the team at BMW was to explore the requested features and also to have some prototype demonstrating their vision. This is what our visualization can do. It shows that for productive interaction with the Visualization GUI, features as described in I.1 of section 7 might need to be implemented. However, our prototype shows the vision and can be demonstrated in a charming way with our Gamification approach (sec. 4.3).

Even without deployment in Kubernetes the cloud supports twenty robots sending coordinate updates every second and chunk updates every 5 seconds as evaluated in sec. 6.1. For much more robots the system should probably by deploying our containers in Kubernetes.

We think that this project was very interesting and we learned a lot. We would like to thank the BMW team for the cool topic and the insights we were granted into their research.

Knowing how hard it can be, to start off with an already existing project, we hope to make it easier for possible successors with this comprehensive documentation. We also added remarks for the development and IDE configuration in the appendix.

A Comprehensive API Documentation

The cloud component offers its API over MQTT and HTTP (REST API). HTTP is preferred for synchronous communication while MQTT is favored for asynchronous tasks.

[GET], **[POST]** and **[DELETE]** refer to REST API requests. **[MQTT]** refers to a mqtt topic.

A.1 Chunk Service

The chunk service stores map chunks and allows to query them. Chunks are uniquely identified by their `row`, `column` and `timestamp` attributes. They are stored either as Strings that are pgm B64 encoded and compressed. Or as B64 encoded png strings.

A.1.1 Meta Data

The endpoint `/api/chunk-service/meta` supports multiple HTTP methods. It is used to globally determine the number of rows, columns etc. of the Map.

[GET] /api/chunk-service/meta Returns the stored `meta.json`. This has typically the following form:

```
{
    "comment":"# CREATOR: GIMP PNM Filter Version 1.1",
    "width":4800,
    "CHUNK_HEIGHT":100,
    "max_brightness":"255",
    "CHUNK_ROWS":32,
    "CHUNK_COLS":48,
    "CHUNK_WIDTH":100,
    "height":3200
}
```

Listing 9: Response by GET /api/chunk-service/meta

[POST] /api/chunk-service/meta This endpoint is used by the robots to store a `meta.json` file. To do this, a robot first checks if there is one available already. If no `meta.json` is stored, it POSTS/PUTS its own version to this endpoint.

[DELETE] /api/chunk-service/meta This endpoint is mainly used for administrative purposes and deletes an existing `meta.json`.

A.1.2 Chunk REST Interface

[POST] /api/chunk-service/chunks Accepts a raw body formatted in `json/application`. Example:

```
{
  "pgmB64compressedData": "eNrt1rERwyAQRUG36HLcjlukABzIAbKPSAcDmt...",
  "row": 3,
  "column": 2,
  "timestamp": "2018-12-20T23:34:33",
  "robotId": "lkjlkj"
}
```

Listing 10: Example JSON body for POST /api/chunk-service/chunks

Responds a json like this:

```
{
  "_index": "chunks",
  "_type": "chunk",
  "_id": "row:3_column:2_2018-12-20T23:34:33",
  "_version": 1,
  "result": "created",
  "forced_refresh": true,
  "_shards": {
    "total": 2,
    "successful": 1,
    "failed": 0
  },
  "_seq_no": 0,
  "_primary_term": 15
}
```

Listing 11: Example JSON response by POST /api/chunk-service/chunks

[GET] /api/chunk-service/chunks This endpoint lets you query chunks. It accepts the following query string parameters:

- `row` (number)
- `column` (number)
- `timestamp` (utc iso string)
- `updatedSince` (utc iso string) returns only chunks with timestamps younger than the given iso string
- `updatedBefore` (utc iso string) returns only chunks with timestamps older than thr given iso string
- `sort` (string) defaults to `'timestamp:desc'`, also supported: `timestamp:asc`, `row:asc`, etc.
- `latest` (boolean) returns only the latest chunk for each combination of `row` and `column` **defaults to false**
- `reduced` (boolean) **defaults to false**, reduces the amount of data returned by omitting the attributes `chunk` and `type` in the results
- `type` (string) **defaults to pgm_base64_compressed** can also be set to `png` to return png base64 strings in the chunk attribute

- `limit` (number) limits the maximum number of returned chunks, **defaults to 10**
- `skip` (number) allows pagination, being combined with `limit`, **defaults to 0**
- `robotId` (string)

Every filter parameter is optional. Therefore, it is possible to query all chunks of one row, for example.

Note: Combining the `latest` and the `updatedBefore` flag returns the complete map at the specified point in time.

The endpoint returns an array of chunks:

```
[
  {
    "row": 3,
    "column": 2,
    "timestamp": "2018-12-20T23:34:33",
    "chunk": "eNrt1rERwyAQRUG36HLcjlukABzIAbKPSAcDmt...",
    "type": "pgm_base64_compressed",
    "id": "row:3_column:2_2018-12-20T23:34:33",
    "robotId": "lkjlkj"
  },
  {
    "row": 3,
    "column": 2,
    "timestamp": "2018-12-14T23:34:33",
    "chunk": "eNrt1rERwyAQRUG36HLcjlukABzIAbKPSAcDmt...",
    "type": "pgm_base64_compressed",
    "id": "row:3_column:2_2018-12-14T23:34:33",
    "robotId": "lkjlkj"
  }
]
```

Listing 12: Example response by GET /api/chunk-service/chunks

[DELETE] /api/chunk-service/chunks Allows to delete the chunk database.

[GET] /api/chunk-service/chunks/id/{id} Returns the chunk information for one chunk. The id will be returned as `_id` in the creation response.

Example:

```
{
  "row": 3,
  "column": 2,
  "timestamp": "2018-12-20T23:34:33",
  "chunk": "eNrt1rERwyAQRUG36HLcjlukABzIAbKPSAcDmt...",
  "type": "pgm_base64_compressed",
  "id": "row:3_column:2_2018-12-20T23:34:33",
  "robotId": "lkjlkj"
}
```

Listing 13: Example response by GET /api/chunk-service/chunks/id/id

This endpoint is used by the *robots* to access the map chunk data in the same format that is also given by the robots. The endpoint should be used by the robots to download the chunks of the entire map on startup.

Also, the *frontend* can query the history of chunks here.

[GET] /api/chunk-service/chunks/id/{id}/png Returns the respective chunk as png.

This endpoint is used by the frontend clients to access PNG formatted chunks that at a certain index. The endpoint should be used by the frontend client to download the chunks of the entire map on startup.

However: if the frontend could process the pgm B64 compressed format on its own, this endpoint could be removed.

A.1.3 Chunk MQTT Interface

[MQTT] iot/chunks/create This endpoint is used by robots to create new chunks. The endpoint expects json strings in the following form:

```
{
  "pgmB64compressedData": "eNrt1rERwyAQRUG36HLcjlukABzIAbKPSAcDmt...",
  "row": 3,
  "column": 2,
  "timestamp": "2018-12-20T23:34:33",
  "robotId": "lkjlkj"
}
```

Listing 14: Example JSON message string to send via MQTT iot/chunks/create

The `chunk` field contains a DEFLATE compressed binary string that is encoded using base64. The `msg_id` is unique for every message while the `robotId` can be used to identify a particular robot.

[MQTT] iot/chunks/advertisement This endpoint is used by the cloud component to advertise new map chunks to frontend clients or robots.

```
{
  "row": 3,
  "column": 2,
  "robotId": "robotId",
  "chunk": "pgm_base_64_encoded_compressed_chunk",
  "png": "png_base_64_string",
  "timestamp": "2018-12-20T23:34:33",
  "id": "row:3_column:2_2018-12-20T23:34:33",
  "chunkUrl": "/api/chunk-service/chunks/
               id/row:3_column:2_2018-12-20T23:34:33",
  "pngUrl": "/api/chunk-service/chunks/id
               /row:3_column:2_2018-12-20T23:34:33/png"
}
```

Listing 15: Example JSON message broadcast by MQTT iot/chunks/advertisement

The `chunkUrl` points to the REST API endpoint returning information about the chunk. Add a `/png` to the string and it returns a png file.

A.2 Coordinate Service

Robots publish their current information via MQTT (topic `iot/coordinates`). The coordinate service listens to this topic and stores the information in elastic search. The data is queryable via the REST API.

[MQTT] iot/coordinates/{robotId} The MQTT topic accepts messages of the following form:

```
{
  "robotId": "blabla",
  "timestamp": "2018-12-20T23:34:33",
  "pose": {
    "position": {
      "x": 10,
      "y": 20,
      "z": 0
    },
    "orientation": {
      "x": 0,
      "y": 0,
      "z": 0,
      "w": 1
    }
  },
  "status": {
    "battery": {
      "battery_level": 3,
      "charging": "sdf"
    },
    "loaded": "dfs"
  }
}
```

Listing 16: Example JSON message string to send via MQTT iot/coordinates/{robotId}

[GET] /api/coordinate-service/robots This endpoint lets you query robot information. It accepts the following query string parameters:

- `robotId` (string)
- `timestamp` (utc iso string)
- `updatedSince` (utc iso string) returns only chunks with timestamps younger than the given iso string
- `updatedBefore` (utc iso string) returns only chunks with timestamps older than thr given iso string
- `sort` (string) defaults to `'timestamp:desc'`, also supported: `timestamp:asc`, `row:asc`, etc.
- `latest` (boolean) returns only the latest information for each robotId **defaults to false**
- `limit` (number) limits the maximum number of returned chunks, **defaults to 10**

- `skip` (number) allows pagination, being combined with `limit`, **defaults to 0**

Every filter parameter is optional.

Note: Combining the `latest` and the `updatedBefore` flag returns the position of each robot at the specified point in time.

The endpoint returns an array with robot data:

```
[
  {
    "robotId": "blabla",
    "timestamp": "2018-12-20T23:34:33",
    "pose": {
      "position": {
        "x": 10,
        "y": 20,
        "z": 0
      },
      "orientation": {
        "x": 30,
        "y": 40,
        "z": 0,
        "w": 0
      }
    },
    "status": {
      "battery": {
        "battery_level": 3,
        "charging": "sdf"
      },
      "loaded": "dfs"
    },
    "_id": "robotId:blabla_2018-12-20T23:34:33"
  }
]
```

Listing 17: Example JSON response by GET /api/coordinate-service/robots

A.3 API Configuration

You can configure the following values by manipulating the process environment values. The following values are the defaults.

- `REST_PORT = 8080` port of this application
- `MQTT = 'test.mosquitto.org'`
- `ES_HOST = '127.0.0.1'`
- `ES_PORT = 9200`
- `SHARE_PREFIX = '$share/bmw/'` this is only needed for a vernmq cluster
- `CREATE_CHUNK_TOPIC = 'iot/chunks/create'`

- `PUBLISH_CHUNK_TOPIC = 'iot/chunks/publish'`
- `NEW_CHUNK_ADVERTISEMENT_TOPIC = 'iot/chunks/advertisement'`
- `COORDINATE_TOPIC = "iot/coordinates/#"`
- `PUBLISH_COORDINATE_TOPIC = "iot/coordinates-frontend"`

A.4 API Development

A.4.1 npm link

The directory `./common` contains the functionality that is required by both the chunk-service an the coordinate-service. It implements a node module that can be required if it is installed. For development we make use of `npm link` because this reflects any changes made in the common folder immediatelly in the other projects. For more information on npm link see here.

So, for development in the chunk-service run the following commands. Same applies for the coordinate-service.

```
$   cd api/common
$   npm install
$   sudo npm link
$   cd ../chunk-service
$   npm install
$   npm link common-api-files
```

Listing 18: Installing the dependencies of the chunk-service

A.4.2 Linting

If you are using Visual Studio Code, it is recommended to install the extension *ESLint* (dbaeumer.vscode-eslint). Made sure to install the node dependencies as described in the section above and open one Visual Studio Code window for each directory: chunk-service, coordinate-service, common

A.4.3 Testing the chunk service

You can run tests for the chunk service as follows:

1. Run elasticsearch and the mqtt broker with the following script:

```
$   cd ..
$   ./run-communication.sh
```

Listing 19: How to only run elasticsearch and mqtt in docker.

2. Open a new terminal and install the dependencies, if not already done:

```
$   cd api/common
$   npm install
$   sudo npm link
```

```
$   cd ../chunk-service
$   npm install
$   npm link common-api-files
```

Listing 20: Installing the dependencies of the chunk-service

3. Run `npm run test`

B Visualization Setup for Development

B.1 Working with Vue.js

- Install required npm packages `npm install`
- Run it locally on port 8080 (hot-reloads changes) `npm run serve`
- Compile and minify for production `npm run build`
- Run your tests `npm run test`
- Lint and fix files `npm run lint`

B.2 Configuration

Configure the application: See Configuration Reference.

Customize api urls: You have three possibilities to configure the ip addresses and ports of the api.

1. **Environment variables:** Set the `API_URL = 'http://ip-address:port'` and the `MQTT_URL = 'localhost:9001/mqtt'` variables
2. **Config file:** add a file named `config.js` to the `visualization` folder. This file should look like this `json { "API_URL": "http://localhost:8081", "MQTT_URL": "localhost:9001/mqtt" }`
3. **Do nothing:** If neither of the above configurations applies, the defaults will be used: `API_URL = 'http://localhost:8081'` and the `MQTT_URL = 'localhost:9001/mqtt'`

B.3 Configuring the IDE

B.3.1 Installation

- Download the VisualStudio Code IDE at `https://code.visualstudio.com/`
- Download Node.js at `https://nodejs.org/`. Node.js comes with npm package manager. This project is originally based on Node.js 10 LTS, e.g. v10.14.2
- Install the following VSCode plugins:
  - ESLint (dbaeumer.vscode-eslint)
  - EditorConfig for VS Code (editorconfig.editorconfig)
  - Vetur (octref.vetur)
  - Docker (peterjausovec.vscode-docker)
  - npm (eg2.vscode-npm-script)
- Run `npm install` to download modules.
- Run `npm run serve` to start the page locally with hot-deploy enabled (automatically reloads once changes are saved).

B.3.2 IDE Settings

1. Open the file settings.json and copy the content (Ctrl.+A, Ctrl.+C)
2. Open VSCode Options: Navigate File > Preferences > Settings
3. Display in raw JSON: Click top right {} element "Open Settings (JSON)"
4. Replace right side "User Settings" with the settings content and save (Ctrl.+V, Ctrl.+S). Can also be put into "Workspace Settings" if the corresponding folder is already opened (Step 5).
5. Open the visualization workspace folder: File > Open Folder, open `cloud/visualization/` This folder contains the package.json, where npm installs the required modules from.
6. Check if autoformat works: Open a `.vue` or `.js`, add unnecessary tabs at the start of a line and save (Ctrl.+S). The line should move back to its indentation level.

B.3.3 Breakpoints in VSCode

If you want to set breakpoints in VSCode, have a look at the extension Debugger for Chrome.

B.3.4 Vue.js Guides

Vue progressive framework for building client-side web applications.

- https://vuejs.org/v2/guide/
- https://vuex.vuejs.org/