DEFO DM2017

Developing a Deformation Monitoring System with the Microservices Pattern

Bachelor Thesis

Martin Andreas Disch

June 2018

Thesis supervisors:

Prof. Dr. Jacques Pasquier-Rocha
and
Pascal Gremaud
Software Engineering Group

UNI FR
UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Software Engineering Group
Department of Informatics
University of Fribourg (Switzerland)

Software Engineering Group

Acknowledgments

I would like to thank Professor Jacques Pasquier and Pascal Gremaud for graciously taking me on despite my project being a bit different from what they usually get. They have been very supportive, while giving me the space I needed to get things done.

I am also very much indebted to GEOINFO Vermessungen AG and Rico Breu in particular for giving me the opportunity to work on an interesting problem, and Tobias Nüssli for always taking the time when I needed feedback. It was a pleasure to work with you.

Finally, I want to thank my family for all the support I received throughout the years.

Abstract

In the construction business it is often necessary to keep an eye on structures or parts of the environment which may move inadvertently. This is done by continuously measuring the positions of several points in the area of interest. The author was given the opportunity to work at GEOINFO Vermessungen AG in Gossau, which provides this service to clients. The company used to collect this data, analyze it manually and upload it to a website where customers can look at it. The goal of this thesis was to develop a web-based system that would analyze the measurements for critical deviations, alert staff if necessary, and automatically transfer it to the existing customer website. Besides automating this workflow, the system should also be easy to extend, reuse, and integrate into other applications. To accomplish this, the microservices architectural style was chosen for the implementation.

Keywords: Microservices, REST, Geomonitoring, Deformation Monitoring, Flask

Table of Contents

List of Figures

List of Tables

Listings

1 Introduction

1.1. Motivation

In the summer of 2017, the author was given the opportunity to work for GEOINFO Vermessungen AG in Gossau. One of the services the company provides is geomonitoring or more specifically deformation monitoring, where infrastructure or terrain is monitored for undesirable movements due to construction activity or other forces. The automatically collected data is then analyzed for dangerous trends and transformed in such a way that it can be uploaded to a different system, which aggregates the data so customers can see it on a website.

While the measurements were automated and continuous, the subsequent processing and transfer of the data was mostly a manual process, aided only by a small set of tools. It took up a lot of time and the results were only available to the customer with some delay.

1.2. Goal

To increase efficiency and relieve engineers from the sometimes arduous task of going through all new measurements manually, a new system was requested. Its primary functionality would be to analyze new measurements and alert staff of noteworthy values, while at the same time transforming and uploading the data directly to the system responsible for the customer-facing website. It would be called DEFO DM2017, an acronym for **Defo**rmation **D**ata **M**anagement and the year it was conceived.

1.3. Organization

Introduction

Contains the motivation and goals of this work, a short summary of the structure of each chapter as well as the formatting conventions used.

Context

A more in-depth look at the problem, status quo and requirements for the final solution.

Technology

Describes constraints that were given, such as the server infrastructure, along with discussion of the choices that were made with respect to the web framework, the high-level architecture of the application, the library for creating basic HTML user interfaces and the way API endpoints were documented.

Services

The application was developed with the microservices pattern in mind. This chapter dedicates a section to each of the eight fundamental services and explains how they work and how they are used in the combined application.

Scheduler

The scheduler is a separate component using the services of the application to execute operations at the time they were scheduled by the user. It performs average analysis on the measurements to detect deviations and uploads measurements for the customers to see. This chapter explains both of these fundamental operations as well as how Python's logging module is used to ensure transparency throughout these processes.

Testing

A brief overview of how unit and integration tests were designed to work for both "normal" code and the endpoints that, by design, are only used by means of HTTP requests.

Conclusion

A review with an evaluation of whether the initial goals were reached, key findings from the project and a discussion of future work on the platform.

Appendix

Contains additional information on how the application can be deployed simultaneously in debug and production environments, the service manual that was written for use at GEOINFO and references that were used throughout this report.

1.4. Notations and Conventions

- Formatting conventions:
  - Abbreviations and acronyms as follows: Application Programming Interface (API) for the first usage and API for any further usage;
  - Code is formatted as follows:

```python
def fib(n):
    """Return the nth number of the fibonacci sequence.

    Parameters
    ----------
    n : int
        The index of the number in the fibonacci sequence

    Returns
    -------
    int
        The nth number of the fibonacci sequence
    """
    return fib_acc(n, 0, 1)


def fib_acc(n, acc1, acc2):
    if n == 0:
        return acc1
    return fib_acc(n - 1, acc2, acc1 + acc2)
```

    Python code is documented according to the NumPy/SciPy standard [5] or in the case of the endpoints (Flask views) according to the OpenAPI specification [10].
- Figures, Tables and Listings are numbered inside a chapter. For example, a reference to Figure j of Chapter i will be noted *Figure i.j.*

2
Context

2.1. Deformation Monitoring

The notion of geomonitoring was already briefly introduced in the introduction. Geomonitoring by itself is a relatively broad term, describing the idea of monitoring the properties of terrain or structures. These properties can include

- The angle of a surface
- Pressure
- Vibrations
- Temperature
- Position

GEOINFO offers solutions for all of these and more [7], but the focus of this project was deformation monitoring. Deformation monitoring is concerned with the position of objects, which can change in some circumstances, mainly due to construction or environmental conditions. The position is measured by placing down small reflectors, referred to as points, and using a total station to measure the distance and angle to them with a modulated wave of electro-magnetic energy that is reflected back to the emitter. See Figure 2.1 for examples. Usually, many points are used in a single project and measured continuously in intervals of some minutes. Once configured, a total station knows the intended positions of all its points and is capable of measuring them autonomously. This technique is used mostly in bridge, dam, landslide, tunnel and construction monitoring. The interested reader is referred to a brochure by Leica Geosystems [8] for some illustrations of these cases.

(a) Reflector (b) Total station

Fig. 2.1.: Reflector and total station for monitoring a railway

Another frequent application of this technology for GEOINFO is the monitoring of railway tracks for construction projects or environmental conditions. It is possible that the tracks start to bend slightly due to the ground underneath them shifting. Because it is essential to anticipate this, the tracks themselves are monitored in these cases by fixing many points to them.

2.2. Status Quo

The total stations for deformation monitoring are produced by Leica Geosystems. They are intended for use with the GeoMoS suite, which integrates the hardware with a software solution. Its main components are a server with a database for all points and measurements and a control software to remotely operate the total stations. The communication between the server and the total stations can run over virtually any medium, most often the cellular network is used for maximum independence of existing infrastructure. All the measurements from the total stations are sent back to the server, where they are stored in the database. Applications can then access this data by running Structured Query Language (SQL) queries on the server.

The next phase after new measurements were logged in the database is analysis. To somewhat facilitate this, a special Excel sheet with Visual Basic for Applications (VBA) scripting was available, which made the right query to access measurements newer than some specified date from the desired project and list them. Additionally, it used coloring to visually highlight the severity of deviations from the target position. Having listed the

unverified measurements, an engineer then had to go over them and determine whether deviations were serious or acceptable. There are many factors going into that decision. For one, different projects have different tolerances. It may be acceptable to have deviations of up to 30 millimeters per day on a hillside, whereas the same amount on a railway track would be devastating. Besides these obvious project specifics, there are also unpredictable environmental factors. There are mechanical ones, for example a construction worker may brush against a reflector by accident and move it slightly. In this case, the engineer would see that only one point is affected and contact people on site to determine the cause of the problem. Or maybe the metal pole to which the total station is attached is bending slightly in the noon sun due to temperature changes in the material. In this instance, a slight deviation in the same direction would be visible for all points, another harmless case. Then, there are the more subtle ones. Hot air rising from the ground can affect the measured distance, direction and angle, as can fog or even rain. Interpreting the results requires experience and knowledge of the specific sites that are monitored, which is why it would be hard to completely eliminate the human element from this step.

After the measurements have been verified and erroneous ones discarded, they can be uploaded for the customer to see. The system responsible for the customer website expects new measurements as Comma Separated Values (CSV) files on a specific network drive. The file names are made up of the project number and the datetime of the upload and the individual measurements contain the point name, datetime of the measurement, measured easting, northing and height in that order, as shown in Listing 2.1.

```
315;22.02.2018 13:00;;731499.6006;276695.1786;499.7456;;
317;23.02.2018 11:31;;731485.4005;276707.3941;502.4275;;
318;23.02.2018 11:32;;731486.1156;276711.2502;500.4581;;
319;23.02.2018 11:32;;731487.3362;276714.7715;498.0493;;
421;23.02.2018 11:33;;731497.5419;276699.2550;499.3213;;
```

List. 2.1: Excerpt from 1460400064.4_20180223110000.txt

The Excel sheet helped with this step as well. With the click of a button, the desired CSV file was created and was then manually moved into the upload directory of the Common Internet File System (CIFS) share that is regularly checked for new files by the customer website.

Fig. 2.2.: The customer website showing points on a railway line

2.3. Requirements

While reliably automating the analysis of measurements for all the special cases mentioned previously would be a bachelor thesis on its own, it is quite clear that the rest of the workflow has potential for optimization. The principal ideas were that the new system could do some preliminary analysis of measurements and determine serious deviations according to a configuration specific to a site. With this done, it could then automatically upload the legitimately safe measurements for the customers to see, while alerting engineers to the others. They would then manually decide whether the measurements in question are likely faulty and should be discarded or are correct and indeed a problem that the customer should be aware of and uploaded. The old and this new workflow are both illustrated in Figure 2.3. Previously, an engineer used the Excel sheet to produce a CSV file they subsequently had to move into the staging directory themselves. Using the new system, the engineer should only have to interact with it to take action on deviations. Everything else runs automatically.

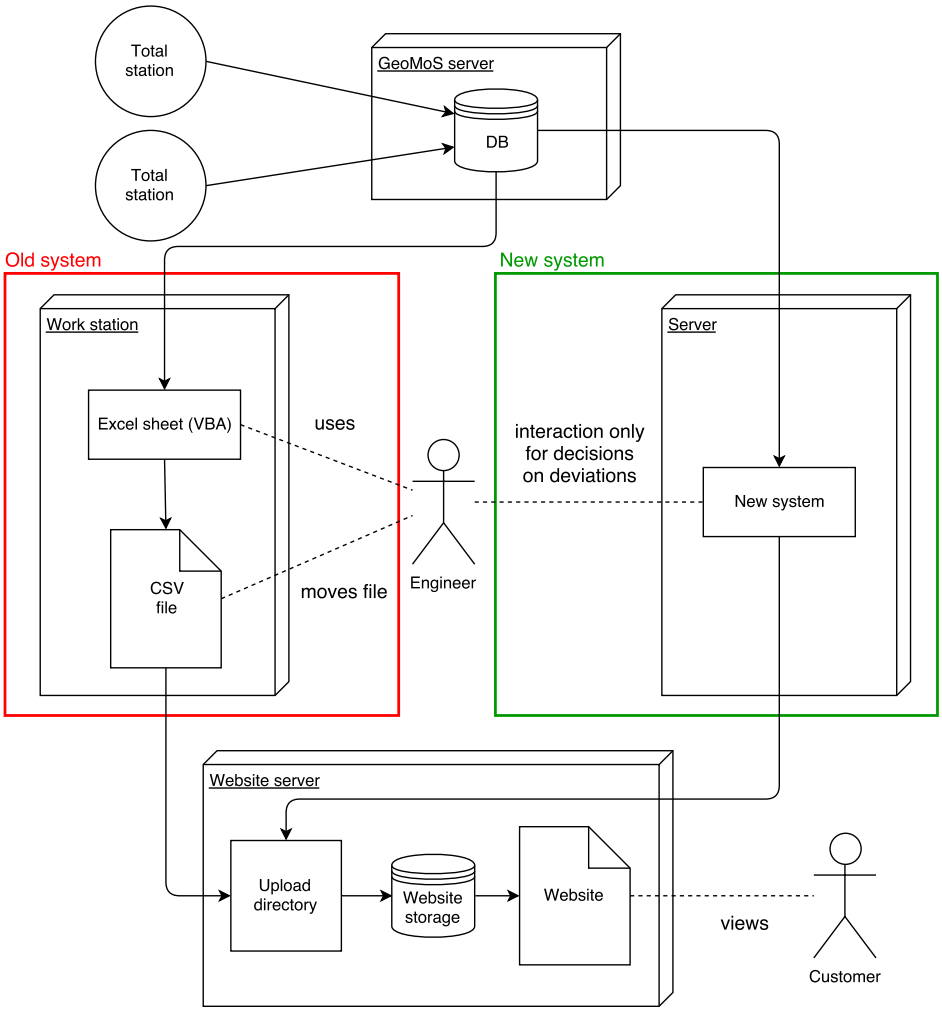


Fig. 2.3.: Data analysis and transfer with previous and new system

A technical requirement was that the system should be web-based, so it could be used without installation on any machines in the company network. Additionally, there were non-functional requirements, such as extensibility, reusability and good documentation. With only these requirements set, the rest was left open and to the judgment of the developer.

To fulfill these requirements, the basic idea was to develop the application with the microservices pattern. Opinions on what this entails vary wildly. Some definitions focus on hard metrics, as in "Microservices are self-contained software components that are no more than 100 lines of code" or "A microservice is an independent software component that takes no more than one iteration to build and deploy" [3]. It is easy to see that these strong definitions do not translate well into the real world for most scenarios. Others give more practical definitions, like "Microservices are independently deployable modules" [4] (translated from German) or that microservices are small, autonomous, and focus on doing one thing well [2]. Purists might argue that weak definitions like these does not live up to the core philosophy of microservices, but the general consensus seems to be that contrary to a monolithic design, such an application is split up into many smaller and loosely coupled services [11]. This modular approach does not only make development and maintenance easier, but it also allows services to be used independently of each other and integrated into other applications. And that was precisely the goal, to allow parts of

the application to be reused or integrated in any potential future development of related software. This architecture also lends itself well to the problem at hand, because it is easy to divide into the different services. There would need to be one concerned with accessing the database and facilitating the retrieval of measurements from individual points in some time interval, another service to create, configure and manage projects (sites) and so on. Some of these services would provide both User Interface (UI) endpoints, as well as Application Programming Interface (API) endpoints which are used by other services of the system and potentially other applications. As a concrete example, the projects service would naturally have its own UI to manage projects, but this UI will use the same API provided by the projects service that external applications could also use. Thus it is conceivable that in the future a different application will be developed with a different technology stack, that can still provide the exact same functionality as the original system by using the same APIs and integrating that functionality into its own UI.

3
Technology

3.1. Server

A web-based system requires some sort of server infrastructure. As far as hardware demands for the proposed platform go, not much is needed. The amount of expected simultaneous active users is very low, at one or two at the most. What is more is that for the majority of time, there is no user interaction at all, apart from project configuration and decision making about critical deviations. Because the system is only available to employees of GEOINFO, there is no risk of a sudden increase in users and since the system is accessed only inside the Local Area Network (LAN), there is no need for geographically distributed hosting. The workload is not too heavy either. A single project site usually has no more than 100 points and with about 5 sites that have analysis and upload scheduled every 3 hours, doing some calculations on the measurements that have been taken in the past hours does not require much computing power.

Having established these requirements, the IT department prepared a virtual Linux machine with the following specifications.

```
geomos@geox0019
OS: Ubuntu 16.04 xenial
Kernel: x86_64 Linux 4.4.0-97-generic
Shell: bash 4.3.48
CPU: 2x Intel Xeon CPU E5-2690 v3 @ 2.594GHz
RAM: 992MiB
```

The system is available inside the LAN with a static Internet Protocol (IP) address. Being strictly internal and not accessible from the Internet has the additional advantage

that no time had to be spent on securing the application from unauthorized access. If engineers need to use the system remotely, they connect to it from their Citrix virtual desktops, which is the only way to work inside the company network from elsewhere.

3.2. REST

Representational State Transfer (REST) is an architectural style that can be used to create services and to which the developed system adheres for the most part. It specifies several constraints that distinguish it from other approaches. The following are condensed versions of the ones described by Erl in [1].

Client-Server

Enforces a strict separation of concerns between the client (consumer) and the server (service). The client invokes the server by sending a request message to the server, which can then return a response message to the client.

Stateless

Refers to the statelessness of the communication between client and server. All information that either of them needs from the other is contained in the sent messages. There is no concept of a session with implicit knowledge.

Cache

Responses from the server are designated as cacheable or non-cacheable so that performance can be improved by routing requests through a cache component. This is the only principle that was neglected in the system developed here, because it will not be deployed on a scale where this would make a meaningful difference.

Interface/Uniform Contract

There is a consistent, uniform interface that is shared by all services that allows identifying and manipulating resources.

Layered system

The internal architecture of the solution is invisible and irrelevant to the client. It can have multiple layers and contain middleware. All of this makes it easier to realize a distributed architecture.

3.3. Web Framework

To easily develop Hypertext Transfer Protocol (HTTP) endpoints that make up the bulk of the platform, a web framework is needed. There is a plethora of solutions built on top of various programming languages, from PHP, Ruby, JavaScript and Python to C# and even C++. Python was chosen as programming language for several reasons. With its extensive standard library it is possible to do almost anything and most of the time with very little code and in a very pleasant way. Due to its popularity it is also one of the most supported languages out there, with a vast amount of libraries. And in the same vein it is probably the most commonly understood of all programming languages, being taught and used in many fields outside of computer science.

While there are many web framework options for Python, web.py[1], Flask[2] and Django[3] appear to be the most recommended. Web.py boasts extreme simplicity and is without a doubt the easiest way to write a few HTTP endpoints, but does not scale too well beyond small to medium-sized applications. At the other end of the spectrum is Django, which is very much intended for large applications. This comes at the cost of increased complexity however, which does not make it the first choice for smaller projects. Flask seems to be somewhere in the middle. It is still very simple if required and therefore a choice for small applications, but also offers features which make it suitable for even medium-sized to large applications. Based on this, Flask was chosen in the end. Other factors for this decision were the high development pace of the project with many contributors, the relatively big ecosystem of extensions, and the many deployment options with different HTTP servers.

3.4. Flask App Design

One of the nice architectural patterns of Flask are the so-called blueprints. A blueprint is a modular component of a Flask app and on an implementation level a blueprint object is similar to a Flask application object. A blueprint is essentially a set of operations that can be self-contained and registered on an application. Blueprints turned out to be a convenient way to implement the different services in a very modular way, by having a blueprint for every service. As an example, Listing 3.1 contains two simplified operations from the projects service.

```
1  from flask import Blueprint, render_template
2  from defo_dm2017.util import config
3  import json
4
5  projects_blueprint = Blueprint('projects', __name__,
6                                 static_folder='static',
7                                 static_url_path='/projects/static',
8                                 template_folder='templates')
9
10
11 @projects_blueprint.route('/projects/<project_name>', methods=['GET'])
12 def projects_name_get(project_name):
13     projects = config.read('projects')
14     # Return 404 if the project doesn't exist
15     if not project_name in projects:
16         error = {
17             'code': 30,
18             'message': "Dieses Projekt existiert nicht."
19         }
20         return render_template('error_standalone.html', error=error), 404
21     # Render projects page with projects as data
22     return render_template('project.html', project=projects[project_name])
23
24
25 @projects_blueprint.route('/projects/api/<project_name>', methods=['GET'])
26 def projects_api_get(project_name):
```

[1]Website: http://webpy.org/
[2]Website: http://flask.pocoo.org/
[3]Website: https://www.djangoproject.com/

```
27      projects = config.read('projects')
28      if not project_name in projects:
29          return json.dumps(
30              {
31                  'code': 30,
32                  'message': "Dieses Projekt existiert nicht."
33              }
34          ), 404
35      return json.dumps(projects[project_name])
```

List. 3.1: Excerpt from views.py

The docstrings of the two functions are rather long because they adhere to the OpenAPI specification for Flasgger as described in Section 3.6. They have been removed here for simplicity. The function projects_name_get() is the endpoint that renders and returns the Hypertext Markup Language (HTML) UI for the configuration of a specific project and projects_api_get() is an API endpoint that returns the configuration of a project in JavaScript Object Notation (JSON) format to be used elsewhere.

This example shows several properties of Flask blueprints. The endpoints are typically implemented in a views.py file as functions using an annotation to pass information about the Uniform Resource Identifier (URI) they implement, the HTTP method and expected parameters. Besides these functions, a new blueprint object is instantiated with parameters for configuration such as its name, locations of static resources or HTML templates. This blueprint object is imported in the Flask app's initialization and registered as shown in Listing 3.2.

```
1 from flask import Flask
2 from defo_dm2017.database.views import database_blueprint
3 from defo_dm2017.projects.views import projects_blueprint
4
5 app = Flask(__name__)
6
7 app.register_blueprint(database_blueprint)
8 app.register_blueprint(projects_blueprint)
```

List. 3.2: Excerpt from __init.py__

Only the database and projects blueprint are shown for brevity, the others are analogous. This is all the code required for a simple Flask app using blueprints. The file structure of the Flask app looks as follows.

```
defo_dm2017/
├── autocontrol/
├── database/
├── decision/
├── file/
├── manual/
├── notifications/
├── overview/
├── projects/
│   ├── static/
│   │   └── style.css
│   ├── templates/
│   │   ├── delete.html
│   │   ├── error_existing.html
│   │   ├── overview.html
│   │   ├── points.html
│   │   └── project.html
│   └── views.py
├── static/
├── templates/
├── util/
└── __init__.py
```

All directories up to the one of the projects service are also blueprints for services and have roughly the same structure. They all contain a views.py file, a directory static/ for their static resources such as specific stylesheets, and a templates/ directory for the HTML templates used in the UI. There are also a static/ and templates/ directory for global resources and templates that are used in all blueprints, such as a generic error page. Finally there is util/ for utility functions also used globally, as well as the __init.py__ file defining the Flask app.

3.5. HTML UI

Developing the UI was not a focus of this project, but still a necessity. While it would have been an option to just use plain HTML for a purely functional experience, this would have looked and felt rather poorly. Luckily, there are many ways to improve the visuals of basic HTML with little effort. The one that was selected in the end is Material Design Lite (MDL)[4]. It is a lightweight library that uses mainly Cascading Style Sheet (CSS) and a bit of JavaScript to enhance HTML components. Its advantage is that it only uses standard HTML elements, so it is only necessary to use the right CSS classes and include the stylesheet and JavaScript to make an existing page of HTML look nice. Due to relying mainly on CSS, it even degrades relatively well in browsers with JavaScript disabled, which is more than can be said of most similar options.

[4]Website: `https://getmdl.io/`

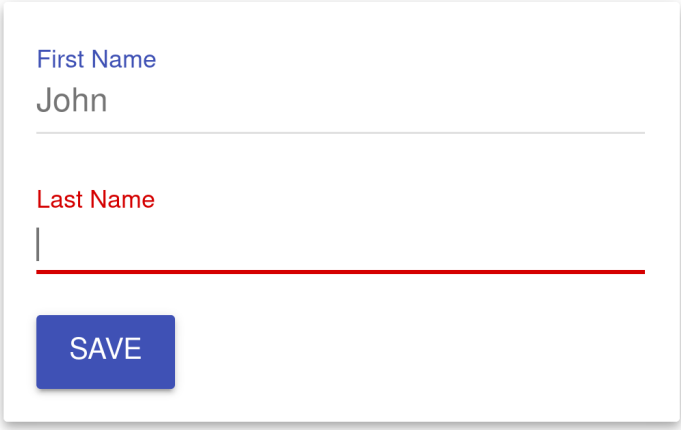


Fig. 3.1.: Simple form with MDL

3.6. Flasgger

Because the final system consists of multiple services, each with its own endpoints, it would be useful to have a way to document all of them during development to stay on top of changes and make sure the documentation is always up-to-date. When developing with Flask, there is already a perfect way to do just that. Flasgger[5] is a Flask extension that extracts OpenAPI specification from Flask views. It integrates easily into a Flask application and uses Swagger UI[6] to display the documentation on `http://localhost/apidocs`. Swagger UI also allows the user to try the endpoints that have been specified by sending HTTP requests and displaying the results from inside the documentation. All of this is generated automatically, as long as the endpoints have docstrings that look like the one shown in the following listing.

```
@database_blueprint.route('/database/api/point', methods=['POST'])
def database_api_point():
    """Return measurements for a single point without daily averages as JSON.
    Builds a query based on the given parameters, logs in to the MS SQL server
    using the credentials stored by /database/config and executes the query.
    If everything went well (200), the result is returned as a list of lists
    (rows) where every row is a point.
    If not (500), a dictionary with the error code & message is returned.
    ---
    tags:
      - database
    parameters:
      - name: db_name
        in: formData
        type: string
        required: true
        description: The name of the database to query
      - name: t_from
        in: formData
        type: string
        required: true
```

[5]GitHub: `https://github.com/rochacbruno/flasgger`
[6]Website: `https://swagger.io/swagger-ui/`

```
22          description: The start time in ISO 8601 format (YYYY-MM-DDThh:mm:ss)
23        - name: t_to
24          in: formData
25          type: string
26          required: false
27          description: The end time in ISO 8601 format (YYYY-MM-DDThh:mm:ss)
28        - name: point_name
29          in: formData
30          type: string
31          required: true
32          description: The point name
33      definitions:
34        point:
35          type: array
36          default: ["7-253R", "28.07.2017 09:05:36", 723669.3963, 246489.6902,
37                    610.6462, 3.3, -1.8, 1.2, 3.7, 3.9]
38        rows:
39          type: array
40          items:
41            $ref: '#/definitions/point'
42        error_db:
43          type: object
44          properties:
45            code:
46              type: integer
47              example: 10
48            message:
49              type: string
50              example: Kein Benutzername/Passwort fuer Datenbank vorhanden.
51                       Einstellungen vornehmen unter /database/config.
52      responses:
53        200:
54          description: The rows (a list of lists) of measurements, where each
55                       measurement is a list consisting of the point's name, date
56                       & time, easting, northing, height, easting diff,
57                       northing diff, height diff, 2D diff and 3D diff.
58          schema:
59            $ref: '#/definitions/rows'
60        500:
61          description: Dictionary with the error code & message
62          schema:
63            $ref: '#/definitions/error_db'
64      """
65      # Code starts here
```

List. 3.3: Docstring for an endpoint according to the OpenAPI spec

This specifies everything, from parameters to accepted data types and possible responses. Swagger UI uses this information for more than just fulfilling its purpose as a documentation. For example, the part of the interface that allows the user to fill in parameters and build HTTP requests to the endpoint will complain if a required parameter has not been filled in. Figure 3.2 shows the overview page of the documentation with a section for every service. The one for the database service is expanded and the different endpoints it exposes can be seen.

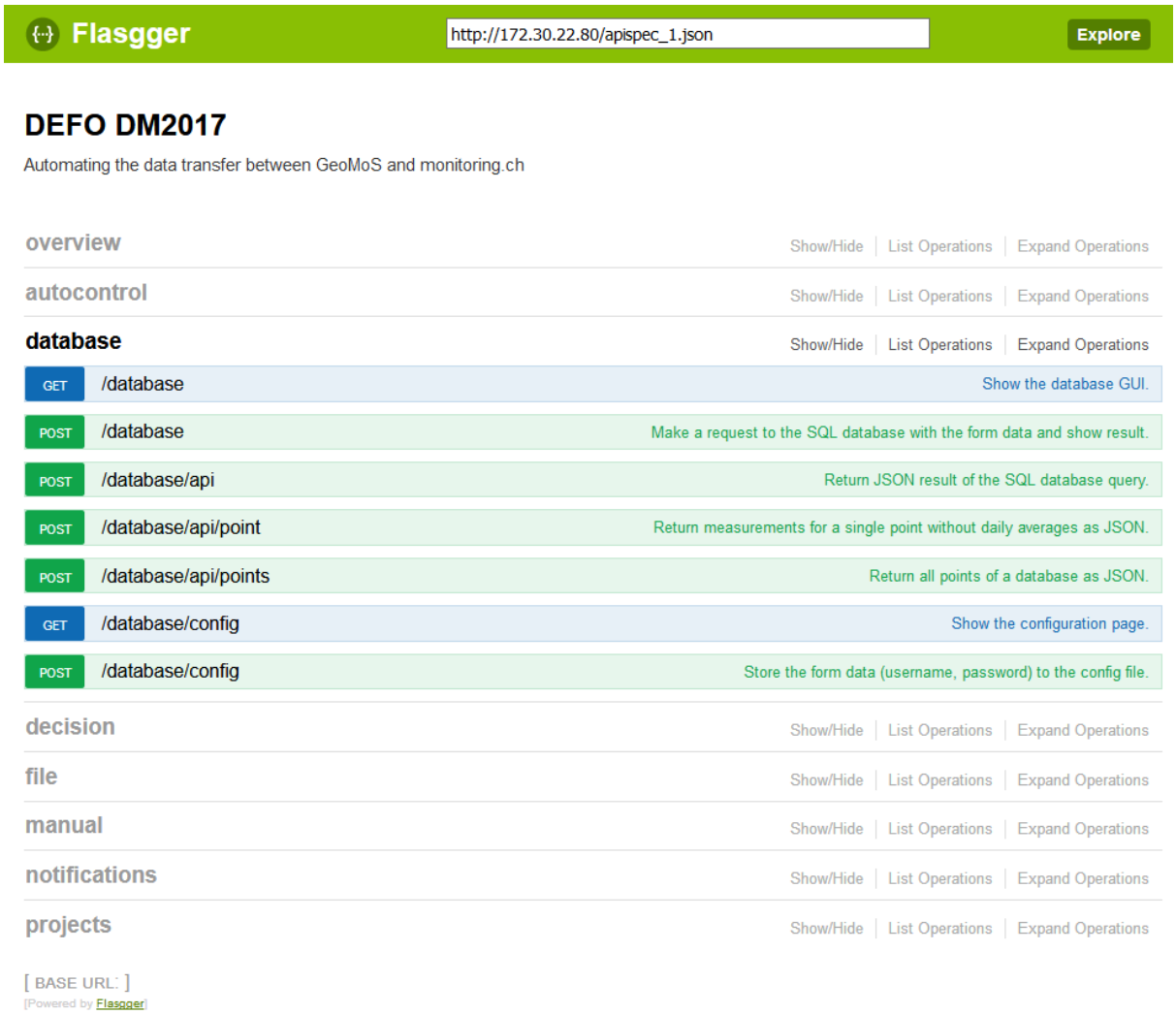


Fig. 3.2.: Endpoint overview of database service in Flasgger

Finally, Figure 3.3 has the detailed overview of a single endpoint within the service with all the data taken from the docstring in Listing 3.3. Note the different parts, such as the endpoint's description, data model and example value of successful responses (HTTP status code 200), detailed specification of the parameter types with inputs to build queries and list of other possible responses such as errors. This way of documenting endpoints has proven invaluable. By keeping the documentation and code together, the likelihood of forgetting to update the docs after changing the implementation can be reduced drastically. That the documentation is always available in the browser is extremely useful as well and the fact that the behavior of the endpoints on different inputs can easily be observed in the same place is icing on the cake. It has helped tremendously to keep track of all components during development and of course it makes it even easier for other developers to continue work on the system or integrate it into different applications.

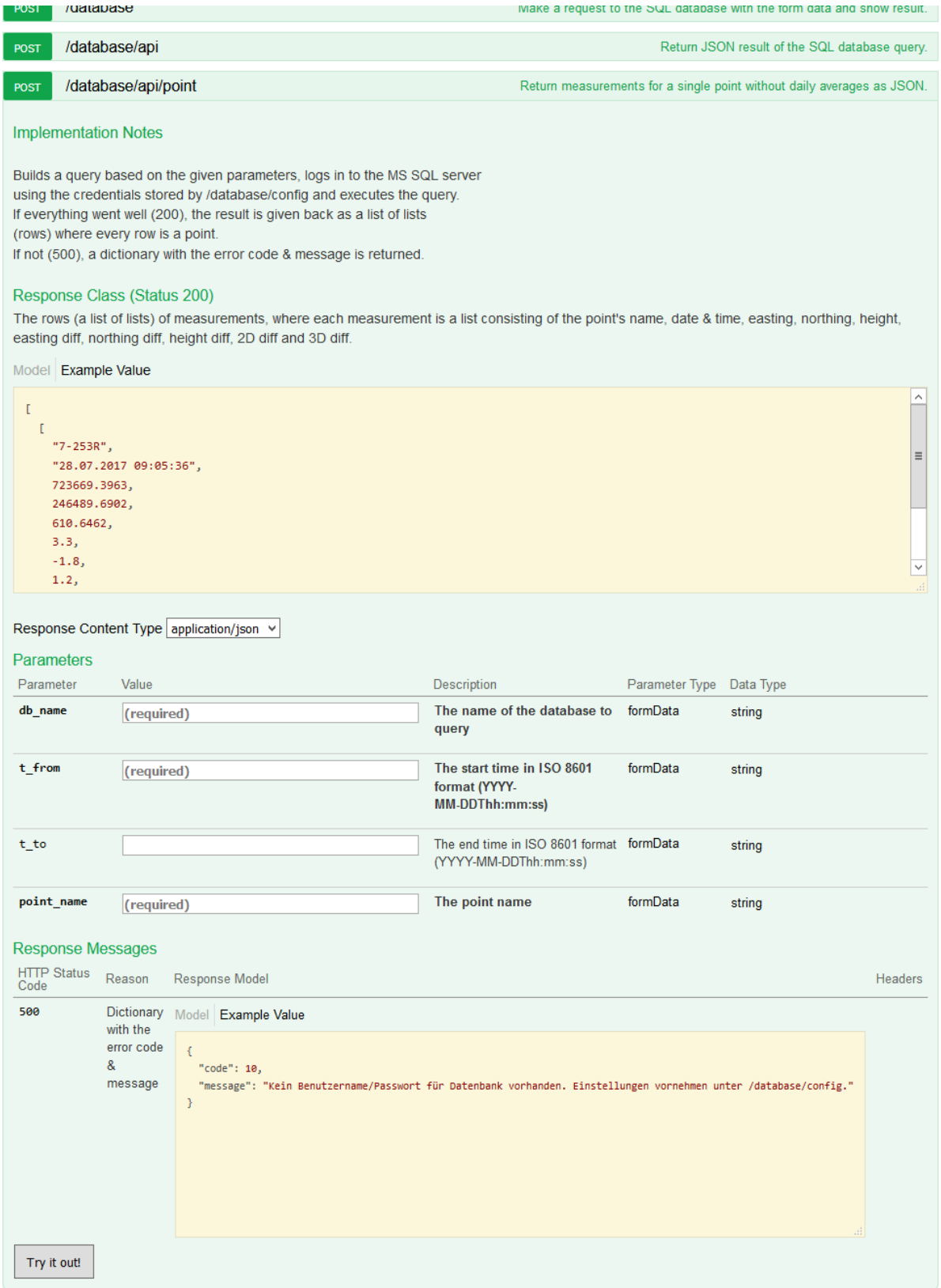


Fig. 3.3.: Overview of a single endpoint of the database service in Flasgger

Because the services have been documented in this standardized way, it is possible to use any implementation of Swagger UI to look at them. Behind the scenes, a small JSON file is generated that contains all the information provided in the docstrings. This is the file

that is loaded by default by the Swagger UI distribution contained in Flasgger, as can be seen at the top of Figure 3.2. This JSON data has been embedded in a newer version of Swagger UI and is made available on the CD of this thesis. The reader is enouraged to take a look at it by opening swagger-ui/index.html in a web browser. It could be useful to see more detailed information about the endpoints when reading the chapter introducing the different services.

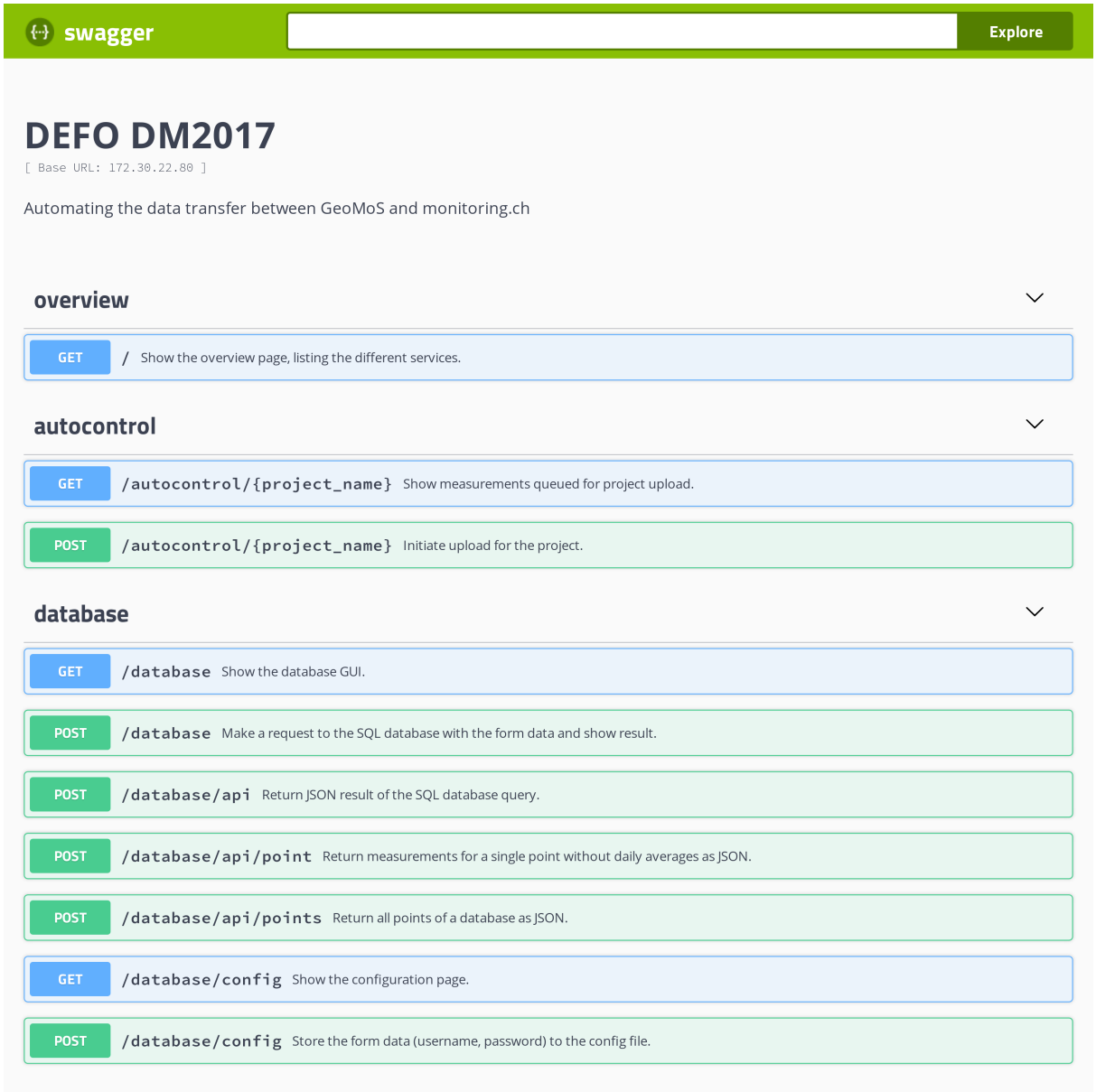


Fig. 3.4.: Swagger UI distribution on the CD

This version of Swagger UI looks slightly different but behaves the same otherwise. Of course it is not possible to send HTTP requests to the endpoints, because they are only available in the company network.

4
Services

4.1. Introduction

This chapter will explore the different services that make up the majority of the system. At the beginning of each section, a table will provide basic information about the endpoints of the discussed service. This is merely there to give a basic overview of the service and does not contain details such as the parameters expected by every endpoint. This information is already present in the endpoint documentation generated from the OpenAPI docstrings and can be viewed using Swagger UI as described in Section 3.6.

4.2. Database

URI	Method	Description
`/database/config`	GET	Shows the configuration page
`/database/config`	POST	Stores submitted configuration data
`/database`	GET	Shows the database UI
`/database`	POST	Shows requested measurements
`/database/api`	POST	Returns requested measurements as JSON
`/database/api/point`	POST	Returns requested measurements for a single point as JSON
`/database/api/points`	POST	Returns a list of all points for a database as JSON

Tab. 4.1.: Database service endpoints

The database service is at the heart of the system and probably the most frequently used in normal operation. Its purpose is to serve as the single point of contact for the rest of the application when the SQL database of GeoMoS needs to be accessed. As such, it is mainly used by other services and the scheduler, but also has some UI components. Those are the first four endpoints listed in Table 4.1 and come in pairs—one to return the HTML code of the page with the forms used to enter configuration properties or make a database request (GET) and one that accepts the posted form data and returns the result page (POST).

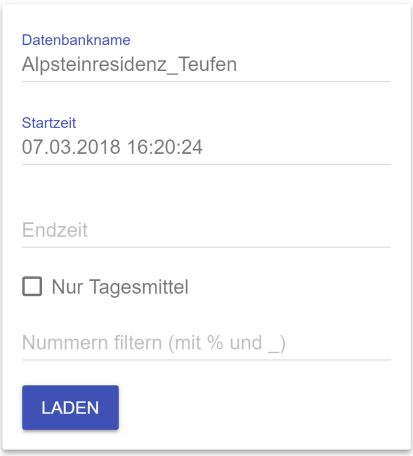
To make requests, the database service needs to log in to the Microsoft SQL server where all points and measurements are stored by GeoMoS. To do that, it needs some credentials which are entered in its configuration page during setup and stored for later use. That is the purpose of the config endpoints and the only thing that needs to be set up to use the database service. Under the hood it is a bit more complicated. To use a Microsoft SQL server from Linux, several utilities are required.

- **FreeTDS**[1] is a collection of libraries allowing applications to communicate with a Microsoft SQL server. It serves as the driver.
- **UnixODBC**[2] is the driver manager and an implementation of the Open Database Connectivity (ODBC) API.
- **pyodbc**[3] is a Python module for connecting to databases using ODBC.

All of these need to be installed during setup, although pyodbc is a dependency of the application package and as such is installed automatically. Some initial configuration is also required to add the data source that is used by FreeTDS and UnixODBC, all of this is explained in detail in the documentation of the system.

Once running, the service can be used by all other services, or even directly by users with the UI endpoints. This is a useful tool if engineers want to look directly at measurements in the database.

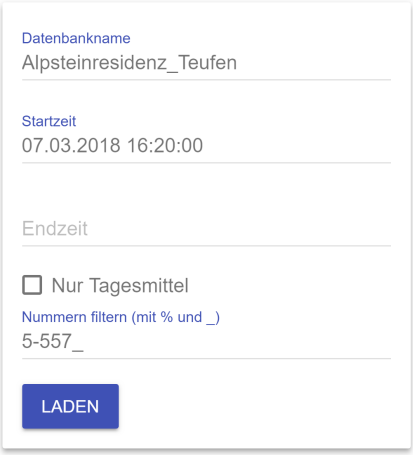
[1] Website: `http://www.freetds.org/`
[2] Website: `http://www.unixodbc.org/`
[3] Website: `https://github.com/mkleehammer/pyodbc`



Nummer	Datum/Zeit	E	N	H	Differenz E	Differenz N	Differenz H
5-556L	08.03.2018 16:19:51	723722.5286	246122.7642	610.6584	8.2	0.3	-7.8
2-314	08.03.2018 16:19:48	731503.6429	276692.5452	498.9046	15.4	13.4	10.0

Fig. 4.1.: Database request UI showing measurements of the last 24 hours

By default, the start time is the current time minus one day and since the end time is optional, this will display all measurements starting from that point. It is also possible to filter for point names, as shown in the following figure. The percentage sign is used to stand in for any number of characters and the underscore for any single character.



Nummer	Datum/Zeit	E	N	H	Differenz E	Differenz N	Differenz H
5-557L	08.03.2018 16:20:55	723723.2706	246117.9363	610.6557	9.0	0.7	-8.3
5-557R	08.03.2018 15:34:31	723721.7667	246117.699	610.6734	9.8	1.1	-3.9

Fig. 4.2.: Database request UI using point name pattern

Figure 4.2 is also a nice example of the specific use case for monitoring railroad tracks. In these cases points are placed in pairs, one for the left rail and one for the right one.

They are named accordingly and by filtering for "5-557_", only measurements for one of these pairs are shown.

This is also a good opportunity to demonstrate another advantage of using MDL for the HTML UI. Whenever user input is accepted, it is good practice to thoroughly check it for mistakes. One of the more delicate ones in this case are the start and end time for looking up measurements. All that is needed to do some preliminary checking of the input and alert the user to their mistakes is to add the pattern attribute to the <input> tag containing a (very long) regular expression to check International Organization for Standardization (ISO) 8601 validity of a date. Should it fail, MDL will nicely display a prepared error message like in Figure 4.3 where a date that does not exist in that year has been entered. Note also that the input labels (e.g. "Nummern filtern") are greyed out and placed inside the input element to save space and are shrunk and moved on top of the text box when something is entered. These floating labels make for a much more visually appealing interface than if they were always present next to the input element.

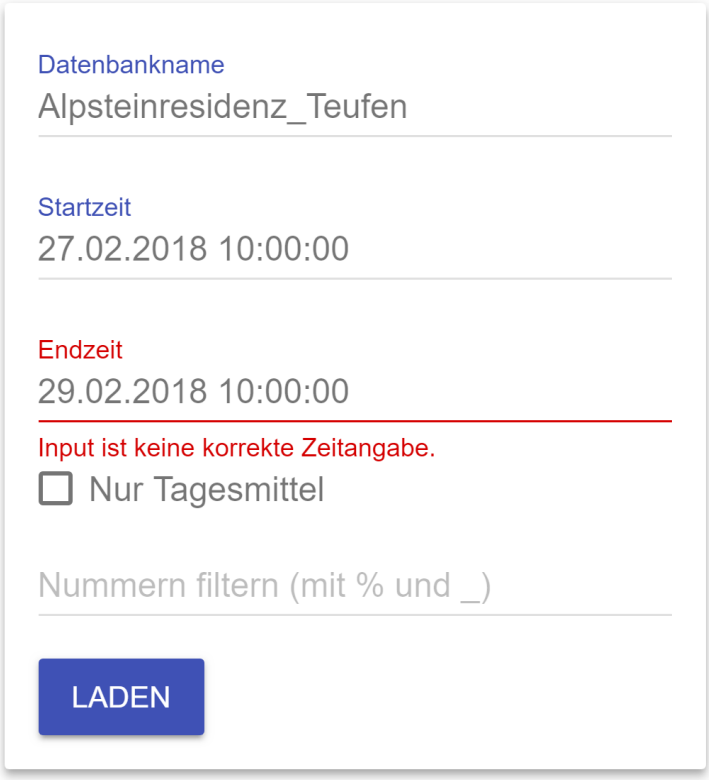


Fig. 4.3.: Database request UI showing error

4.3. Projects

URI	Method	Description
`/projects`	GET	Shows the projects overview page
`/projects`	POST	Creates a new project using submitted information
`/projects/<project_name>`	GET	Shows the configuration page of a project
`/projects/<project_name>/settings`	POST	Stores the submitted project settings
`/projects/<project_name>/points`	GET	Shows the point selection page for a project
`/projects/<project_name>/points`	POST	Updates the project configuration with the submitted points
`/projects/<project_name>/delete`	GET	Shows a page asking the user if they really want to delete this project
`/projects/<project_name>/delete`	POST	Deletes the project based on the submitted decision
`/projects/api/<project_name>`	GET	Returns project properties as JSON
`/projects/api`	GET	Returns configuration of all projects as JSON
`/projects/api/<project_name>/pointupdate`	POST	Updates the settings for an individual point of a project with the submitted data
`/projects/api/<project_name>/autoupdate`	POST	Updates automation data for a project

Tab. 4.2.: Projects service endpoints

The projects service is used to create and manage projects. It also has several endpoints that serve the UI and accept submitted data entered by the user, as well as API endpoints that are used by itself and other services to access that data.

The `/projects` endpoints are relatively simple, because all they do is list the existing projects and allow the user to create a new one. Figure 4.4 shows this page with the existing projects and the input form for new ones.

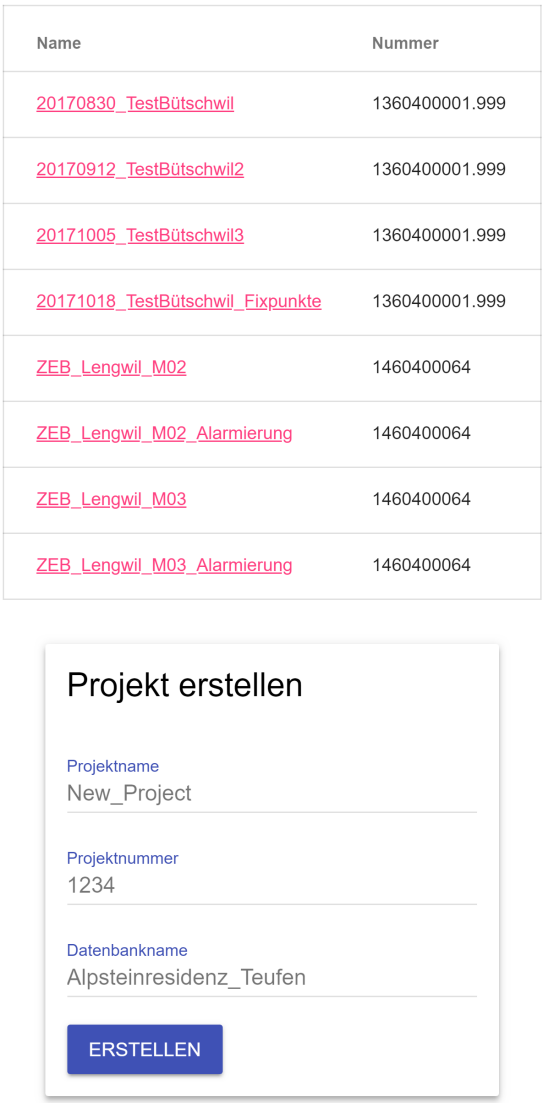


Fig. 4.4.: Projects overview

Once a project has been created, it needs to be configured. That is where `/projects/<project_name>` and and `/projects/<project_name>/settings` come in. The former displays the project settings page shown in Figure 4.5 and the latter accepts the submitted form data from it. This project settings page is rather large, which is why it has been cropped slightly to fit on a single page. It contains settings for two more evaluation techniques besides average analysis ("Mittelwertauswertung"), but since they are not yet implemented, they were cut out to save space. Below the card containing the settings there is usually also a table with all points that have been selected to be part of the project. This is a topic that is further elaborated upon a bit later in this section.

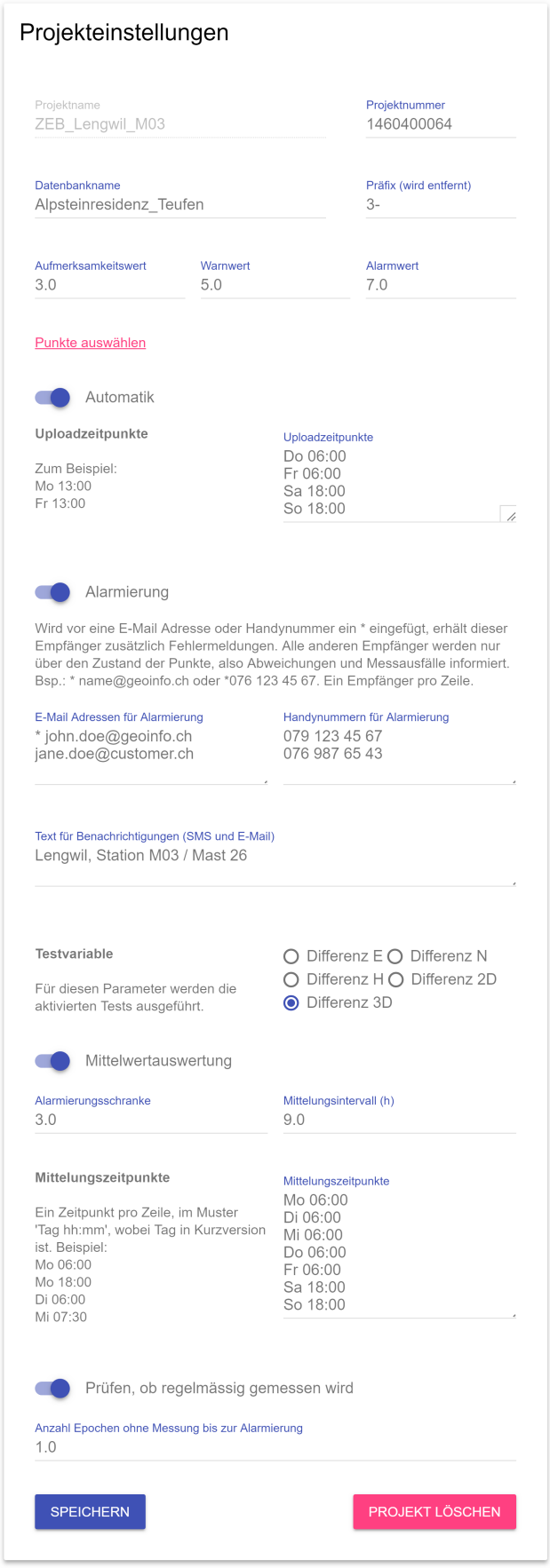


Fig. 4.5.: Individual project settings

The settings page begins with some basic information about the project, such as its name, number and the name of the database where its measurements are stored. It is also possible to enter a prefix that is used in the point names. This option is there because points are sometimes prefixed by the number of the corresponding total station and this information is not used in the website that shows measurements on a map to the customer. Hence this option to enter a prefix, which is removed from all point names during export of the measurements for the customer website.

The next few values define different escalation levels for deviations. They are used to color measurements in the interfaces of other services differently according to the severity of the deviation, which makes it easier to detect patterns in the data.

The link after that leads to a page where points from the database can be selected to add them to the project. This is also discussed later on.

The first switch is used to control whether this project is considered at all in the scheduler which initiates average analysis and exports as planned by the user. It is there as a way to quickly deactivate a project.

The text area below contains the times for uploads. At these points in time the scheduler will initiate an upload for the project, which means that it will take the latest valid (no deviations over threshold) measurement for every point in the project and export this data as a CSV file to the customer website.

The following switch controls whether alarms will be sent for critical events, such as large deviations or error messages. The text areas below contain email addresses and mobile phone numbers which will receive emails or SMS respectively in these cases. This part of the settings is concluded with another text field containing some data that will supplement the alert messages that are sent.

The radio group for the test variable defines what exactly is considered during analysis. The measurements stored in the GeoMoS database have the easting, northing and height of the deviation and any one of them—as well as the two-dimensional and three-dimensional deviation that are calculated based on them—can be selected as the variable to be tested. Most often this is the 3D deviation, because movements in all directions are relevant for almost all projects.

The next section is for the average analysis. It can also be turned off globally for a project. There is a text field for the alert threshold, that is the deviation in millimeters after which a notification is generated. The other input is for the analysis interval, which means that the average of all measurements in that amount of past hours is used to determine the deviation that is checked. After that there is a text area similar to the one for the uploads, but this time defining the times when average analysis should be performed. The scheduler will look at the measurements for all points of the project according to the settings made previously and determine whether measurements are valid and stored for later export or if there are problems for which notifications should be generated.

The final section is used to enable a check of whether there are new measurements at all for the points of the project. It is possible that a point has moved so much that the total station can no longer focus on it. Another scenario is that there is a problem with the total station itself, preventing the taking of new measurements. If this is active, alerts will be sent in case any points have no new measurements in the n last epochs (instances of average analysis).

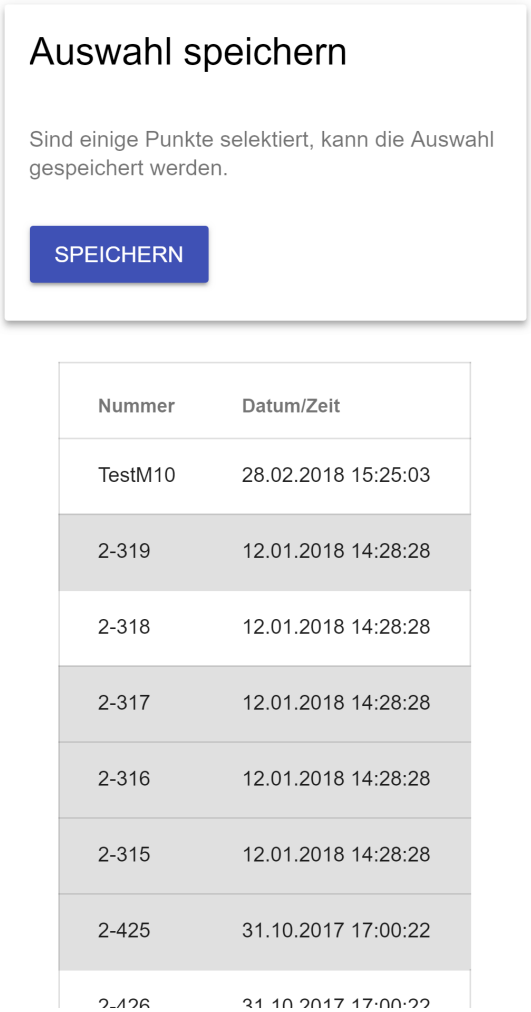


Fig. 4.6.: Point selection for a project

Figure 4.6 shows the point selection that was mentioned previously and is made available by the `/projects/<project_name>/points` endpoint. It lists all available points and lets the user select those that belong to the project. Implementing this selection required some JavaScript trickery. The points are listed in a table and clicking on a cell highlights it if it was not previously active and reverts it to normal if it was. Additionally, the user is able to select a point, hold the shift key and select a different one to activate or deactivate both of them and all points in between. In order to send the current selection to the endpoint in a POST request, the names of the active points are written as a JSON list to a hidden input field inside the form that is sent off on a click of the save button.

This leaves the final part of the projects service UI, the individual point settings. While it is well and good to be able to set project-specific tolerances for deviations and such, there is a need to be able to do that on a per point basis. That is why all points are listed below the projects settings as shown in Figure 4.7.

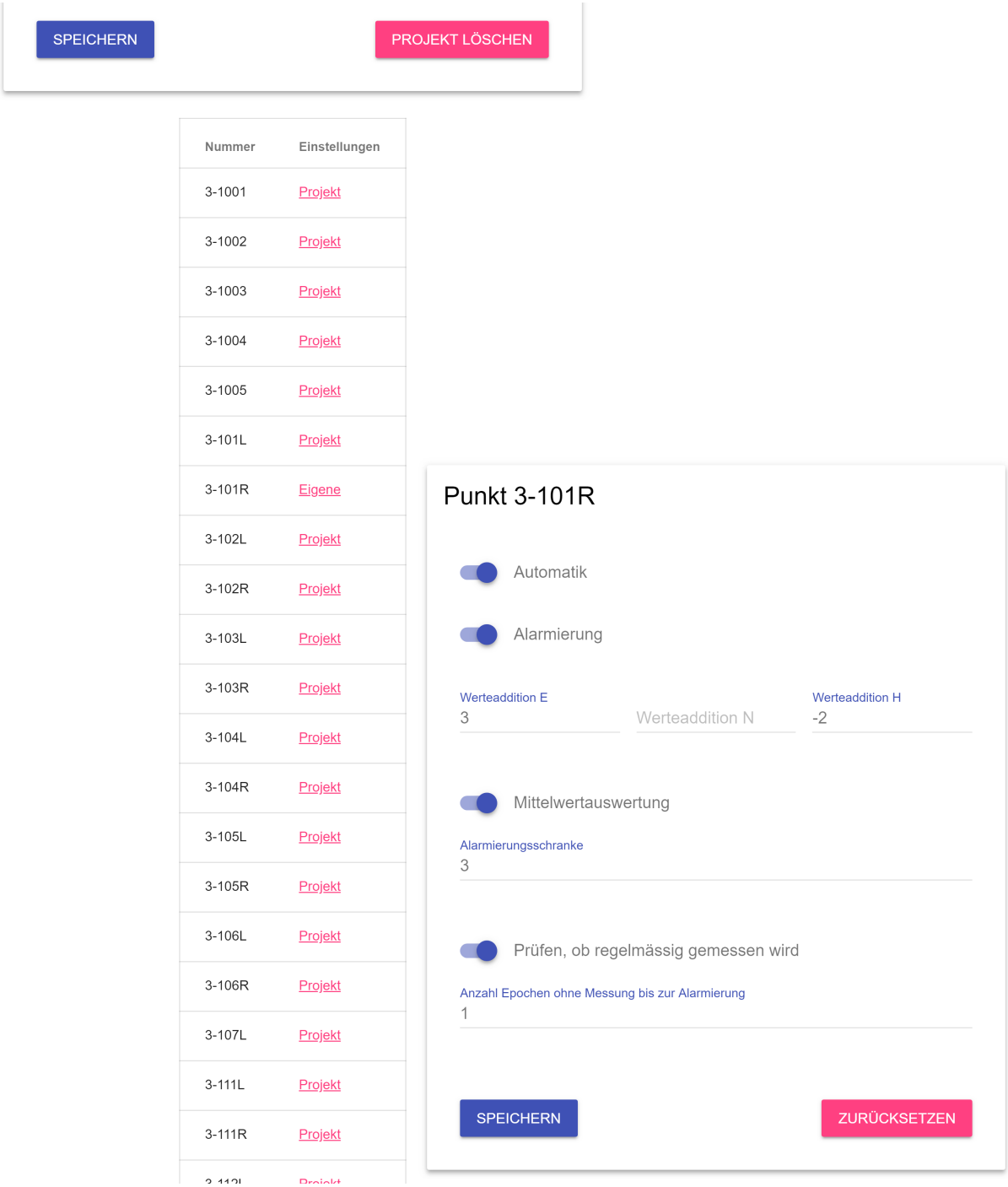


Fig. 4.7.: Individual point settings

By default, points inherit the project settings, as denoted by "Projekt" next to the point name in the table. A click on that brings up a floating window for making settings that apply to a specific point. It is possible to individually deactivate them completely (switch "Automatik"), disable alerts (switch "Alarmierung"), set up value additions and make custom settings for average analysis and the check if measurements have been taken recently. When individual settings have been made for a point, this fact is represented by a changed text in the list. In the screenshot, this is the case for point 3-101R.

The value additions are also worth discussing, because they will become relevant again

later. Their purpose is to reflect the fact that a point may have moved because of some reason and that its new position is still acceptable and should be treated as the new expected placement. For example, if point 3-101R has moved 3 millimeters westwards and 2 millimeters up, its easting deviation would be -3 and the height 2. But because this is the accepted new position, a value addition of 3 for the easting and -2 for the height can be set up, effectively canceling out the deviation, leading to no deviation at all. A different part of the system offers the users a simple one-click way to accept a deviation as the new position, setting up the required value additions which are then visible in this window.

On a technical note, this is the part that required the most effort as far as client-side scripting goes. Until now, all UI components have used forms to submit values from input elements via POST requests to receiving endpoints. To quickly check and make individual point settings, it would not be efficient to have to load a page for every single point and being returned to the overview after saving changes. For that reason, the floating window was implemented. Clicking on one of the point links makes an Asynchronous JavaScript And XML (AJAX) call to the `/projects/api/<project_name>` endpoint to get the project configuration and load either the project defaults, if the point uses those, or its custom settings, if any have been made. The floating window is then made visible in the Document Object Model (DOM) and filled with the right data. Upon saving changes, another AJAX call is made, sending the settings to the `/projects/api/<project_name>/pointupdate` endpoint, which saves them. With this approach, many changes can be made fluently without having to reload the page at all.

This covers the most relevant parts of the projects service. The `/projects/<project_name>/delete` endpoints are only used as an interstitial asking the user to confirm before deleting the project and `/projects/api/<project_name>/autoupdate` is an endpoint designed specifically to store new automation data for a project. Automation data is part of the project configuration that is invisible to the user. It is used to keep track of average analysis and uploads. For example, a record is kept for every point of a project, containing the last valid measurement and the last uploaded measurement. This information is then used by the scheduler. This will be discussed in more detail in Chapter 5.

4.4. Manual

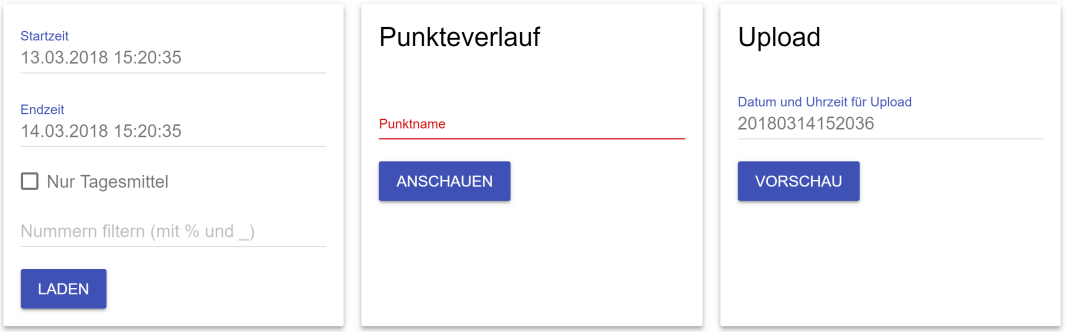
URI	Method	Description
`/manual/<project_name>`	GET	Shows the manual export UI for a project
`/manual/<project_name>`	POST	Shows requested measurements
`/manual/preview`	POST	Shows a nicely formatted export preview
`/manual/export`	POST	Exports the edited measurements to CSV file
`/manual/graph`	POST	Returns a point graph as image/svg+xml data
`/manual/history`	POST	Returns a HTML table with the history of a point

Tab. 4.3.: Manual export service endpoints

The manual export service mainly does what the name implies. It closely resembles the old workflow that was presented in the context chapter. Its purpose was as a proof of

concept, to show that the individual parts of the system work and could be automated on top of that. It also serves as a fallback for engineers to use if they need to upload data to the customer website manually. In short, the service lets engineers load measurements from a time span for a project and applies some color (as per the project settings) to make spotting deviation patterns easier. It is then up to the user to determine, which measurements are most likely faulty and should be excluded from upload. Once they have been identified, the engineer can initiate a manual export, which takes the most recent non-excluded measurements for all points of a project, uses them to create the appropriate CSV data and hands it over to the file service to be written to a file in the right directory on the specified network share, where it is read by and later shown on the customer website.

The GET version of the `/manual/<project_name>` endpoint presents the user with a basic interface where they can set up filters for which measurements to see in the same way as in the database service. These arguments are then submitted to the POST version, which returns the measurements as shown in Figure 4.8.

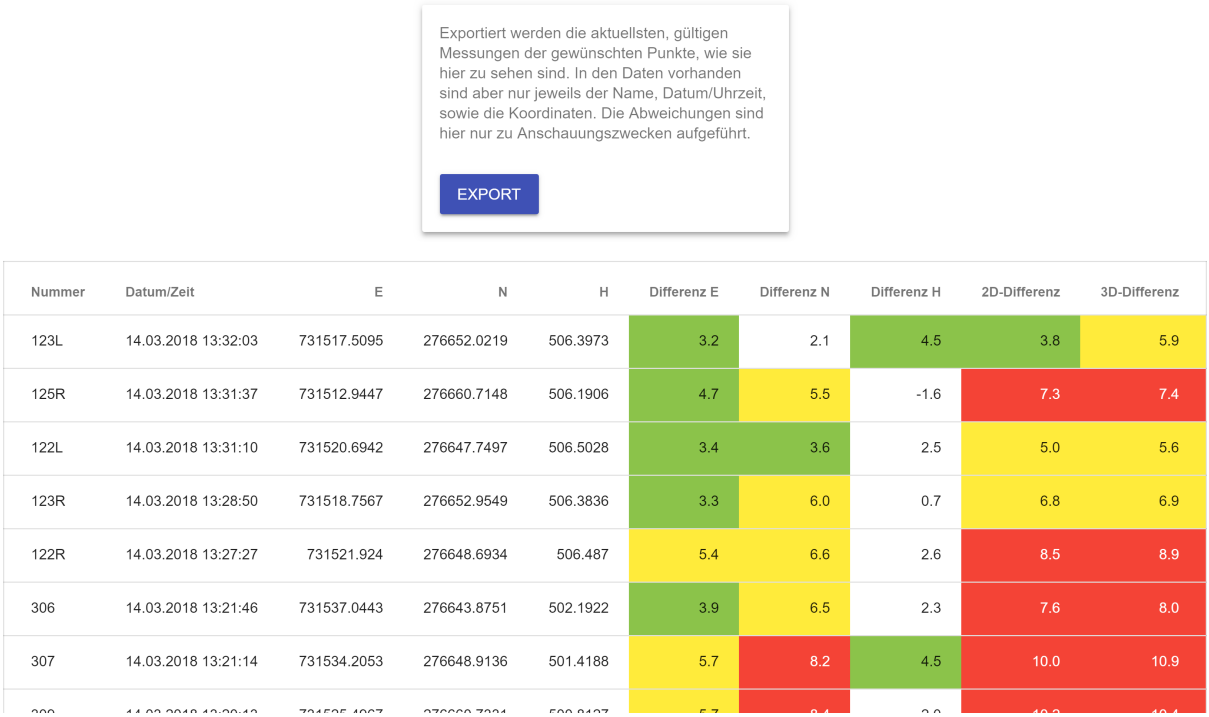


Nummer	Datum/Zeit	E	N	H	Differenz E	Differenz N	Differenz H	2D-Differenz	3D-Differenz
2-127L	14.03.2018 13:34:57	731505.4846	276667.8865	505.995	8.4	8.2	-2.4	11.7	11.9
2-127R	14.03.2018 13:34:28	731506.8821	276668.7251	505.9819	7.3	7.5	-9.0	10.4	13.8
2-126L	14.03.2018 13:33:59	731508.6723	276663.8732	506.0918	7.0	5.0	-4.8	8.7	9.9
2-125L	14.03.2018 13:33:30	731511.7287	276659.763	506.2055	7.4	5.6	-0.5	9.2	9.2
2-126R	14.03.2018 13:32:57	731510.0248	276664.7317	506.0758	8.2	4.7	-12.0	9.4	15.2
2-124L	14.03.2018 13:32:31	731514.5865	276655.9269	506.3024	7.5	6.6	3.6	9.9	10.6
2-123L	14.03.2018 13:32:03	731517.5095	276652.0219	506.3973	3.2	2.1	4.5	3.8	5.9
2-125R	14.03.2018 13:31:37	731512.9447	276660.7148	506.1906	4.7	5.5	-1.6	7.3	7.4

Fig. 4.8.: Measurement exclusion for the manual export

In this example, the six most recent measurements have been determined to be faulty (perhaps due to a slight movement of the total station) and should not be uploaded. Using the same scripted mechanism already mentioned for point selection in the project settings, the user can select one or more measurements. Once they have been marked, a custom date and time that will be used in the CSV file name together with the project number can be entered in the upload card on the top right. For convenience, a default value using the current time is generated when the page is loaded. With a click of the button

for the preview, all necessary data—such as the chosen time interval and possible point name filters, as well as which measurements were excluded—is submitted to the `/manual/preview` endpoint, which generates a preview page where only the measurements that will be exported are shown for confirmation.



Nummer	Datum/Zeit	E	N	H	Differenz E	Differenz N	Differenz H	2D-Differenz	3D-Differenz
123L	14.03.2018 13:32:03	731517.5095	276652.0219	506.3973	3.2	2.1	4.5	3.8	5.9
125R	14.03.2018 13:31:37	731512.9447	276660.7148	506.1906	4.7	5.5	-1.6	7.3	7.4
122L	14.03.2018 13:31:10	731520.6942	276647.7497	506.5028	3.4	3.6	2.5	5.0	5.6
123R	14.03.2018 13:28:50	731518.7567	276652.9549	506.3836	3.3	6.0	0.7	6.8	6.9
122R	14.03.2018 13:27:27	731521.924	276648.6934	506.487	5.4	6.6	2.6	8.5	8.9
306	14.03.2018 13:21:46	731537.0443	276643.8751	502.1922	3.9	6.5	2.3	7.6	8.0
307	14.03.2018 13:21:14	731534.2053	276648.9136	501.4188	5.7	8.2	4.5	10.0	10.9

Fig. 4.9.: Manual export preview

If the most recent measurement of a point has been excluded, the next most recent one that is not is used. In the end, there will be one correct measurement for every point in the project. Note that in the preview the prefix that was set up in the project settings is already removed from the point names to reflect that this will be the case for the CSV export. By a final click on the export button, all of this is submitted to the `/manual/export` endpoint. It generates the CSV data and sends it to the file service, which is responsible for writing it to the correct location to be read into the customer website's storage.

In practice, the manual export is frequently used as a tool to quickly get a glimpse of the state of a specific project. To make it even more useful, engineers asked for a tool to visually observe the history of a point. For this purpose, the point graph was implemented. It is shown by entering the name of the desired point in the middle card. After clicking the button, a new card is dynamically inserted between the top row of control cards and the table of measurements, showing a line graph of the point's measurements for the specified time interval. The graph is interactive, the exact values of a measurement are displayed when the mouse pointer is hovered on top of it and individual data series can be turned on and off.

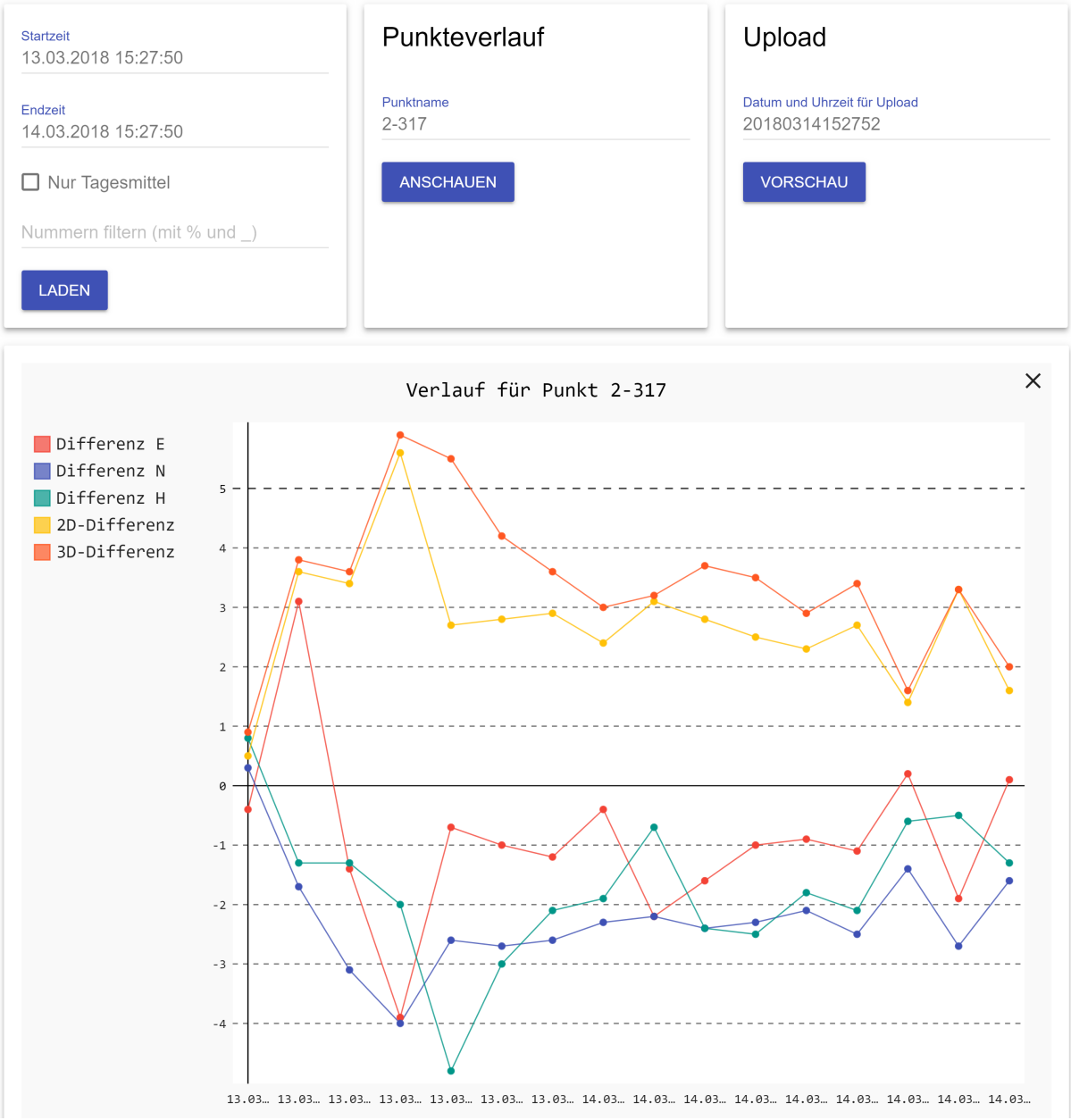


Fig. 4.10.: Manual export point graph

When the user is done with the graph, they can simply click the cross in the top right corner and the card will be removed from the layout without reloading the page. All of this is once again made possible by some client-side JavaScript. In this case, it manipulates the DOM to insert and remove the card and sends an AJAX call to the `/manual/graph` endpoint to receive the graph. This endpoint uses the Pygal[4] library to generate the graph. Pygal can output graphs in many different formats, but here a Base64 encoded data URI (data:image/svg+xml) that is directly embeddable in `<image>` tags is used. Additionally to the graph, a small table showing the measurements in chronological order is also displayed below it. This table is generated by the `/manual/history` endpoint, which is also used in an AJAX call.

[4]Website: `http://pygal.org/en/stable/`

4.5. Notifications

URI	Method	Description
`/notifications`	GET	Shows all notifications
`/notifications`	POST	Shows filtered notifications
`/notifications/config`	GET	Shows configuration page
`/notifications/config`	POST	Stores the submitted configuration
`/notifications/api`	GET	Returns all active notifications as JSON
`/notifications/api/add`	POST	Adds a new notification
`/notifications/api/delete`	POST	Deletes one or more notifications
`/notifications/api/get_id`	GET	Returns a notification ID that will be reserved
`/notifications/api/send`	POST	Sends one or more notifications via email or SMS

Tab. 4.4.: Notifications service endpoints

The notifications service allows other components to create notifications, which can be displayed for the user in a notification center of sorts and sent via email or SMS. The two `/notifications` endpoints show the notification center. By default, this is a chronological listing of all notifications (GET).

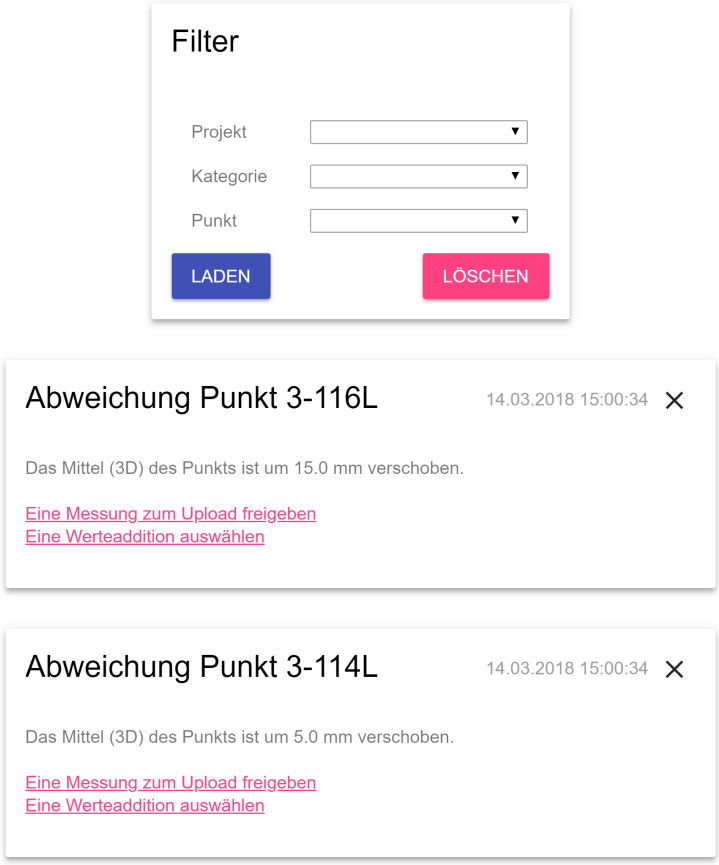


Fig. 4.11.: Notification center

It is also possible to filter notifications according to project, category (e.g. deviation, no measurement) or point name.

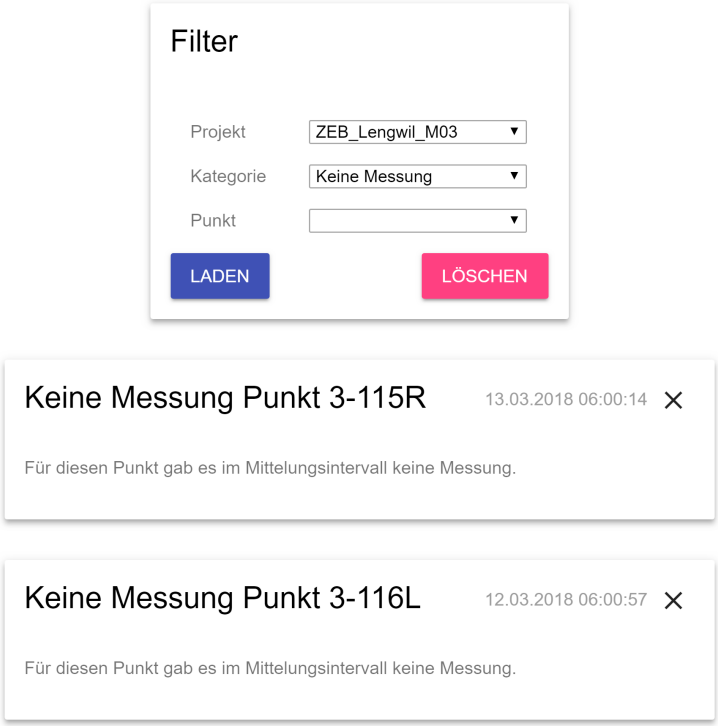


Fig. 4.12.: Notification center using filters

Notifications can be dismissed (deleted) individually with the small cross in the top right corner of a card. This happens dynamically using JavaScript, the card is removed locally from the DOM and an AJAX call is sent to the `/notifications/api/delete` endpoint to delete it on the server side. Notifications can also be deleted in batches by using the delete button in the filter card. In this case, all notifications to which the filter applies (all notifications that are currently shown) are deleted at once.

At this point it might be interesting to look a bit further at how notifications are added with the `/notifications/api/add` endpoint. Notifications are sent as JSON dictionaries containing key-value pairs. Two keys, *title* and *message*, are required. There are also optional keys that are used for filtering.

- *project_name* is the name of the project that caused this notification
- *point_name* is the name of the point in question
- *categories* is a list of categories that apply to this notification

The filter system is rather flexible, as none of these additional properties are required (which would not make sense, after all not all notifications are necessarily related to an individual point). There are also no fixed categories, instead the service users can add categories freely by simply using them in the notifications they post. The category filter will display all categories contained in the current set notifications. Finally, there is another optional notification property. It is called *short* and contains a short version of the notification text. It is used when sending notifications via email or SMS, because for SMS in particular, it is better to be brief.

```
"23352":{
         "point_name":"3-115R",
         "datetime":"13.03.2018 06:00:14",
         "short":"115R: Keine Messung",
         "title":"Keine Messung Punkt 3-115R",
         "message":"Fuer diesen Punkt gab es im Mittelungsintervall keine Messung.",
         "project_name":"ZEB_Lengwil_M03",
         "categories":[
            "Keine Messung"
         ]
      }
```

List. 4.1: JSON notification example

Usually when adding a notification, it is automatically assigned an Identifier (ID). However, there are cases where the ID of the notification has to be known beforehand and it has something to do with the type of notification seen in Figure 4.11. The text of deviation notifications contains two links to actions the user can take. They link to a UI of the decision service, which is introduced in Section 4.6. In short, the user can do something about the notification, after which they are returned to the notification overview. Since the notification has been dealt with, it would be nice if it had automatically been dismissed already so the user can continue looking at others. To do that, the decision service needs to tell the notifications service to delete the notification after the user has acted on it. For that, it needs to know which notification it was that led the user to it in the first place. This information is contained in the two links that are shown in the body of the notification. The links are built as `/decision/{method}/{project_name}/{point_name}/{noti_id}`. All information needed by the decision service is in there, including the notification ID. Since the notification's body with the links was built before sending it to the notifications service, the ID had to be known at that stage. This is what the `/notifications/api/get_id` endpoint is for. It returns a new ID and reserves it for later use. With this ID, the links for the notification can be built and the JSON dictionary making up the notification is sent to the notifications service. The ID is sent as well, to tell the notifications service to use this instead of assigning a new one.

This whole process seems a bit complicated. Would it not be easier to implement the notifications service in such a way that it could be told that a notification is of the deviation type and should contain two links that are then built and added to the text by the notifications service, which knows which ID it is going to assign, instead of the component that creates the notification? It would have been simpler since the `/notifications/api/get_id` endpoint would not be necessary, but would have been more complicated at the same time because then the notifications service would have had to be aware of different types of notifications and how to create links for the decision service. This would have introduced a dependency to another service. There are many cases in the system where dependencies are inevitable and by design. For example, almost all services need information about projects from the projects service to provide their functionality. But in general, it is better to design the services as loosely coupled as possible. And since the notifications service can be built as a completely independent entity, that is how it was done. With the current solution, the notifications service is very much self-contained, only accepting notifications with titles, messages (containing HTML for rich message bodies with links and similar) and some filtering properties. And in case

the notification ID needs to be known beforehand to do something advanced with it, the option is there.

Once notifications have been created, they can also be sent via email or SMS. This is also something the notifications service takes care of. Usage is very simple with the `/notifications/api/send` endpoint. It only needs to know the IDs of notifications that are batched into a single message, optionally a bit of additional text that is sent along and the email addresses and mobile phone numbers of the recipients. To make all of this work, there needs to be a bit of initial configuration of course. The `/notifications/configuration` endpoints do that. For email, the IP address or domain name of a Simple Mail Transfer Protocol (SMTP) server that allows sending email from arbitrary email addresses for whitelisted clients needs to be known, as well as the email address that is used as the sender in messages. SMS is a bit more complicated, because an external provider is needed to send SMS messages. After a period of evaluation, iNetWorx AG[5] was chosen. To use this service, additional authentication information needs to be entered in the configuration page together with a phone number that is used as the sender for messages. Because iNetWorx's SMS gateway is prepaid, users can also set up a threshold for the number of remaining messages. The notifications service uses the iNetWorx API to check the quota after every send operation and issues a notification warning the user to pay for more messages.

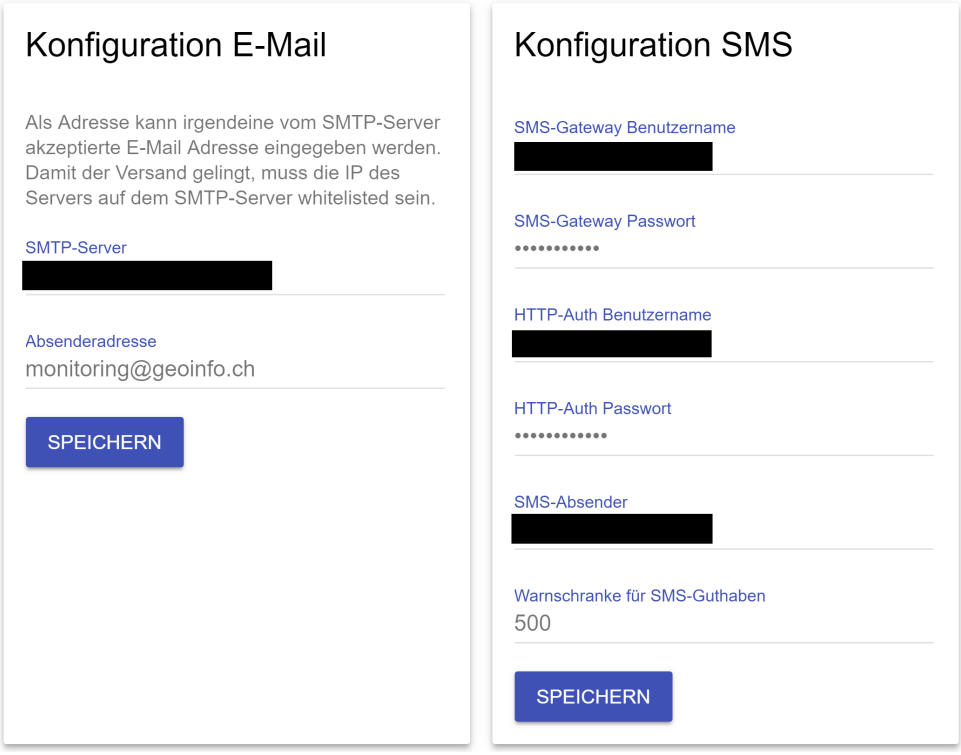


Fig. 4.13.: Notifications service configuration

This notification sending functionality—SMS in particular—is very useful to the company, because there are some projects where customers request to be kept very closely in the loop about the state of their site. If necessary, it is possible to set them up in the project configuration to also receive notifications immediately as they are generated.

[5]Website: `https://www.inetworx.ch/en`

4.6. Decision

URI	Method	Description
`/decision/{method}/{project_name}/{point_name}/{noti_id}`	GET	Shows the decision page
`/decision/{method}/{project_name}/{point_name}/{noti_id}`	POST	Shows the decision page with filtered measurements
`/decision/update`	POST	Saves the selected addition or the alarm addition and releases measurement

Tab. 4.5.: Decision service endpoints

The decision service is accessed by clicking one of the two links inside a deviation notification to decide what to do with this point. When a notification has been issued, this means that the average of the measurements for the point in question is over the defined threshold. This could be due to a real deviation, faulty measurements or because a point has been moved intentionally. Since it is impossible to determine automatically whether the customer should be worried or not, the most recent measurement is not automatically uploaded and a notification generated to alert an engineer. They then take a look at the data and decide what to do. The first link ("select a measurement for upload") presents the engineer with a list of measurements. They can select one of them and manually clear that one for upload. It is then saved in the automation data of the project as the last valid measurement and will be uploaded the next time an upload is scheduled, unless replaced by a more recent valid measurement until then.

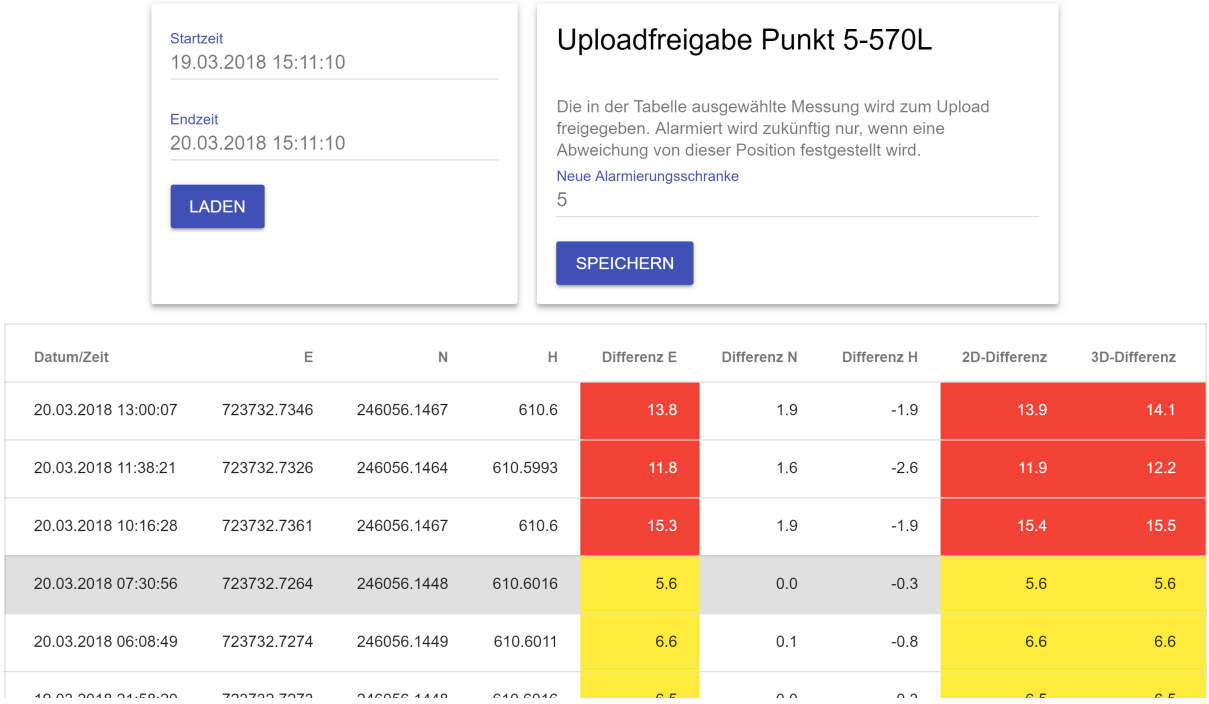
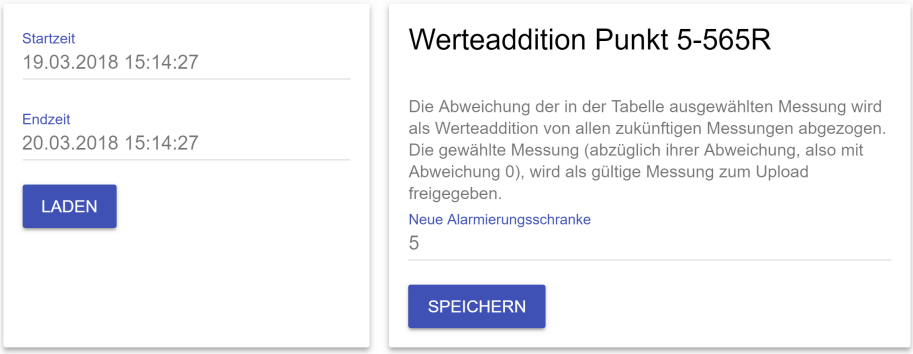


Fig. 4.14.: Decision for measurement upload

This is what would be done in cases where a deviation has been determined to be real and a measurement should indeed be uploaded. Clearing a measurement this way has the additional effect of setting the so-called alarm addition. After a deviation has been validated as being a real deviation instead of just faulty measurements, it would not make sense to continue to generate notifications during every average analysis because the deviation is still there. After all, the engineers have been made aware of the problem and have acknowledged it. From now on, measurements for that point with this deviation are let through and uploaded without causing another notification, because the problem has been verified and the customer should see it. The acknowledged deviation is subtracted (or rather the negative deviation added—hence the term "alarm addition") from new measurements of the point before checking for deviations. This means that roughly same deviations are not recognized as such, which is the goal. But as soon as a significant (greater than the threshold) additional deviation from that position occurs, it is treated like every other deviation: measurements are held back, a notification is generated and an engineer needs to assess the new situation.

The second link ("select a value addition") is a similar but different story. Its purpose is to set the value addition that was introduced in Section 4.3 about the projects service. A value addition is required when a point has been moved manually to a new position. This will show up as a deviation and to correct that, one of the measurements can be selected to serve as the new default position for the point. This also clears the selected measurement for upload, as it does not technically have a deviation anymore.



Datum/Zeit	E	N	H	Differenz E	Differenz N	Differenz H	2D-Differenz	3D-Differenz
20.03.2018 13:02:33	723727.6159	246079.6399	610.6314	9.4	0.9	-0.9	9.5	9.5
20.03.2018 11:40:51	723727.6182	246079.6403	610.6311	11.7	1.3	-1.2	11.8	11.9
20.03.2018 10:18:47	723727.6169	246079.6403	610.6311	10.4	1.3	-1.2	10.5	10.5
20.03.2018 08:57:34	723727.6142	246079.6393	610.6313	7.7	0.3	-1.0	7.7	7.8
20.03.2018 07:33:20	723727.6118	246079.6392	610.6319	5.3	0.2	-0.4	5.3	5.4
20.03.2018 06:12:08	723727.6121	246079.6387	610.6311	5.6	-0.3	-1.2	5.6	5.7
20.03.2018 04:53:14	723727.6109	246079.6387	610.631	4.4	-0.3	-1.3	4.4	4.6

Fig. 4.15.: Decision for value addition

It is worth emphasizing the difference between the alarm addition and the value addition.

Selecting a value addition—deliberately setting a new position for a point—is more significant. It shows up in the point settings and has a greater effect than just setting an alarm addition. With a value addition, the point does no longer have the chosen deviation in the eyes of the system, because it is subtracted before considering the measurements any further. This means that the deviation does not show up in the data that is presented in the manual export or the decision service at all. Only when using the database service to look directly at the measurements, the deviation is still visible. An alarm addition on the other hand only has the effect that measurements will be uploaded despite having the chosen deviation and no notifications will be generated. The deviation is still there when looking at measurements in manual export or when making decisions.

The top right card in both interfaces also has an input for a new alert threshold. This is the same setting that is also present in the project settings and will load the same value. This is something that was requested so engineers have a shortcut for raising the threshold in case it is set too low.

By a click of the save button, all data is posted to the `/decision/update` endpoint, which will perform the selected alarm or value addition, clear the measurement for upload and delete the notification of the deviation that was just dealt with. The user is then redirected back to the notifications page to continue with the next one.

4.7. File

URI	Method	Description
`/file/config`	GET	Shows the configuration page
`/file/config`	POST	Stores submitted configuration
`/file/api`	POST	Writes submitted data to file

Tab. 4.6.: File service endpoints

Whether it is manual upload or the automated version invoked by the scheduler, measurements are exported to the customer website at some point. As was already described, the website expects the CSV data in a file with the project number and a date in the filename, which is to be written to a specific directory on a network share. This is where the file service comes in. It is only responsible for one operation, and that is writing submitted data to a file with a given name located in this directory. Because the path to this directory is a variable that could potentially change in a different setup, it can be configured. This is the only option that can be changed in the configuration endpoints of the file service.

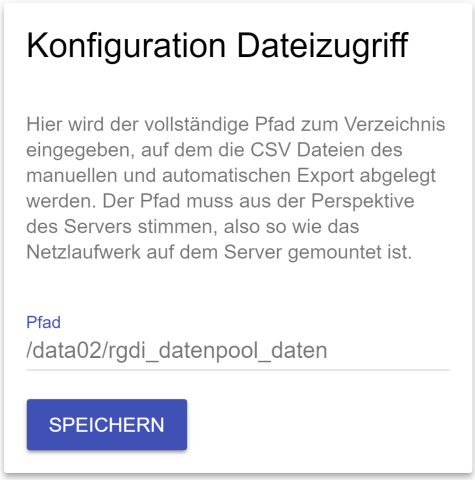


Fig. 4.16.: File service options

With this set up, the file service has everything it needs to do its job. Whenever a manual or automatic export occurs, it will call the `/file/api` endpoint, sending along the prepared CSV data and a filename. The file service will then take care of writing it and returning a status code once it is done.

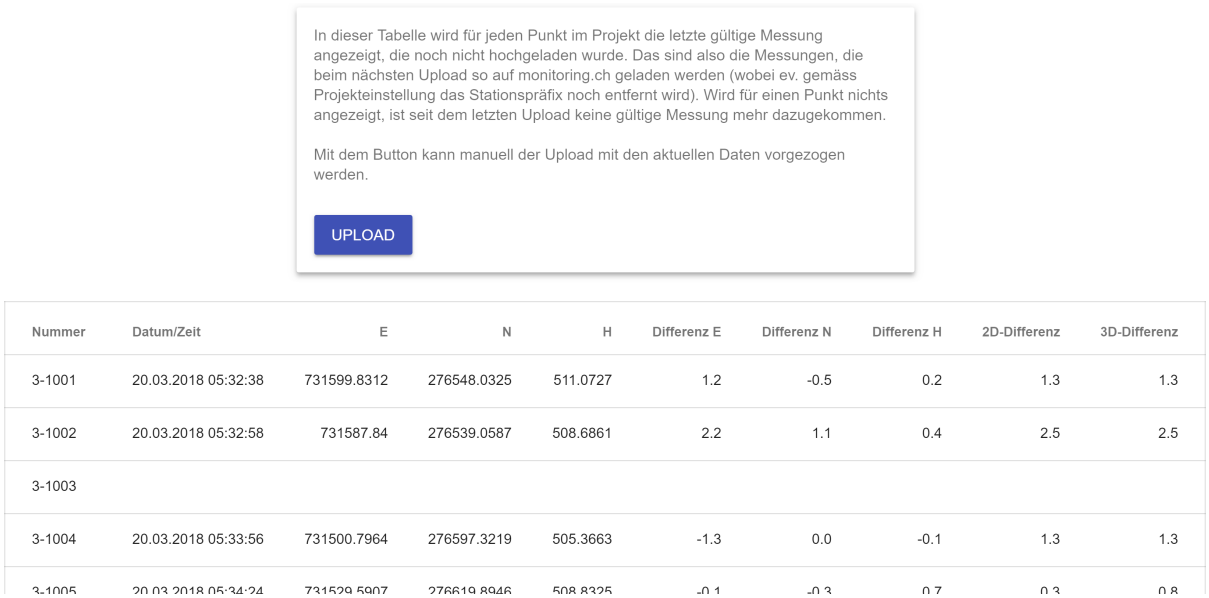
It might seem strange to put so little functionality into its own service, but actually this is exactly what the microservices pattern is all about. Not only that, it also makes perfect sense just in the context of this part of the application. That is because the file writing functionality is required by multiple other application parts (manual and automatic export) and if they each were to write the file themselves, both would need to know the path to the network share, leading to duplication in the configuration.

4.8. Autocontrol

URI	Method	Description
`/autocontrol/{project_name}`	GET	Shows measurements queued for upload
`/autocontrol/{project_name}`	POST	Initiates upload for a project

Tab. 4.7.: Autocontrol service endpoints

The autocontrol service was implemented to make the workings of the system a bit more transparent and give more control over the automated upload to users. It was previously explained that during average analysis, the most recent valid measurement of a point is stored for later upload, which is started by the scheduler according to what the user set up in the project settings. The autocontrol endpoints allow the user to look at these measurements and initiate an early upload manually.



Nummer	Datum/Zeit	E	N	H	Differenz E	Differenz N	Differenz H	2D-Differenz	3D-Differenz
3-1001	20.03.2018 05:32:38	731599.8312	276548.0325	511.0727	1.2	-0.5	0.2	1.3	1.3
3-1002	20.03.2018 05:32:58	731587.84	276539.0587	508.6861	2.2	1.1	0.4	2.5	2.5
3-1003									
3-1004	20.03.2018 05:33:56	731500.7964	276597.3219	505.3663	-1.3	0.0	-0.1	1.3	1.3
3-1005	20.03.2018 05:34:24	731529.5907	276619.8946	508.8325	-0.1	-0.3	0.7	0.3	0.8

Fig. 4.17.: Autocontrol service

Figure 4.17 shows this page where the user can see all points of the current project, as well as the last valid measurements. This is exactly the data that is uploaded, any potential value additions are of course already done, as explained in Section 4.6. In case there is no valid measurement for a point that has not already been uploaded, it is shown blank. This is the case for point 3-1003. Here, a notification has been generated when a deviation was detected, but has not yet been acted upon. Since the cause of the deviation is unclear, nothing is uploaded for this point until an engineer has cleared it.

By clicking on the upload button, the upload is started manually. After succeeding, these stored last valid measurements for all points of the project will be flushed and remain blank until an average analysis yields more recent valid measurements that have not already been uploaded. Internally, not only the last valid measurement, but also the last uploaded measurement is stored in the automation data of every point, in order to prevent duplicate uploads.

4.9. Overview

URI	Method	Description
/	GET	Shows the overview page, listing all services and documentation

Tab. 4.8.: Overview service endpoints

By now, all services, which provide all of the user interface and most of the application logic have been introduced. What is missing is something tying them all together, to make them usable like a single and coherent application. For this, yet another service was created, although it barely deserves this name. It has a single GET endpoint and uses this to serve an overview page.

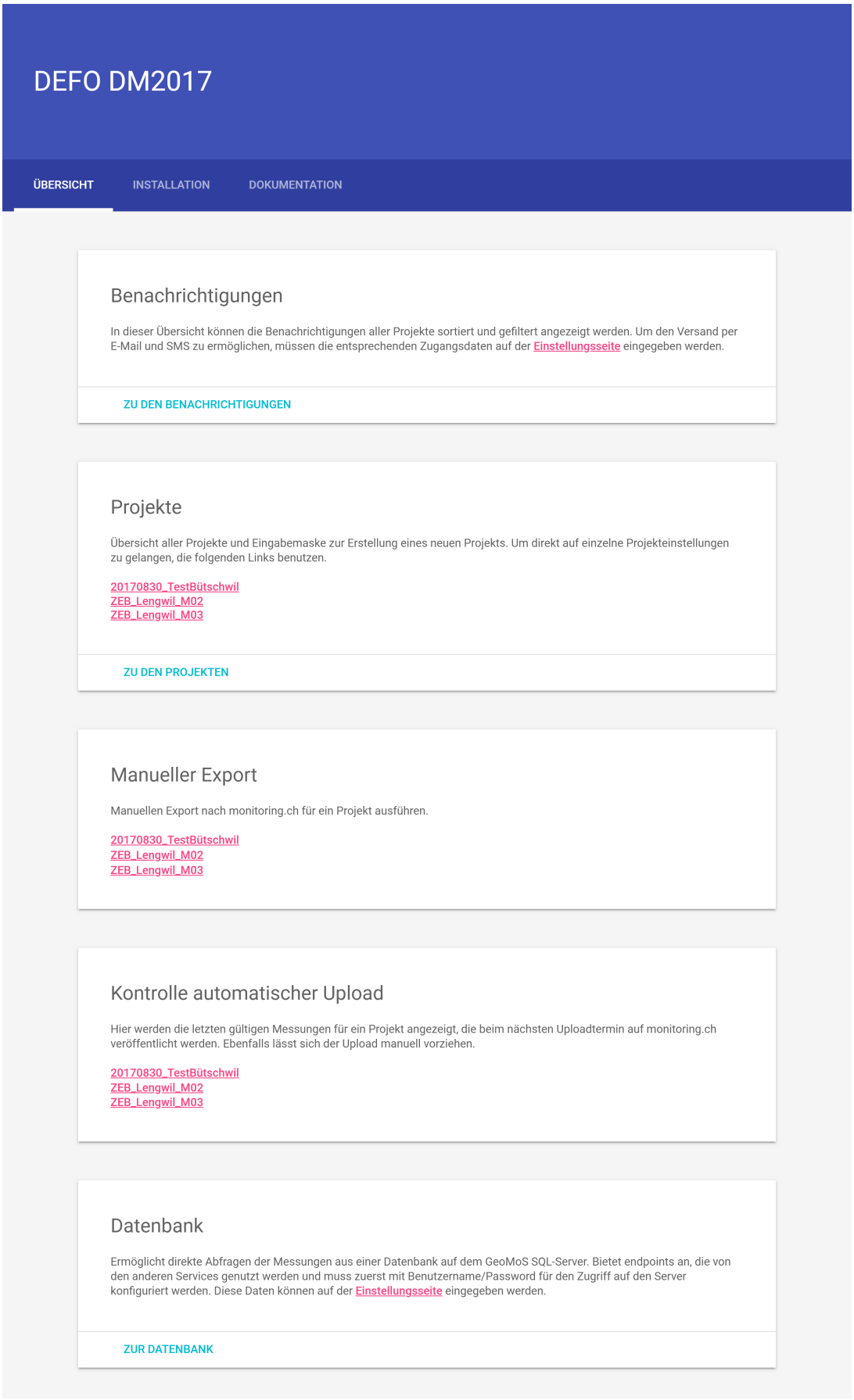


Fig. 4.18.: Overview page

The page contains three tabs, the default being the service overview. This serves as a quick entry into the parts of the application that users might be interested in, ordered by how frequently they are used.

The first card is for the notification service. It has a link to quickly go to the notification center, because once projects have been set up, this is where engineers will spend most of their time. The second one is for the projects service. Besides providing a link to the projects overview page, this card will also list all projects with links to go directly to the individual settings page of a project for faster access. The third card does the same thing for the manual export by providing links that will lead directly to the manual export interface for the chosen projects. The fourth is another variation of the same principle, this time for the autocontrol service where users can gain more insight into the upload process and initiate it manually. Finally, in case users want to access the database directly using a convenient interface, they can do so by following the link in the card for the database service.

The second tab lists useful information and links to configuration pages that are used for first-time setup.

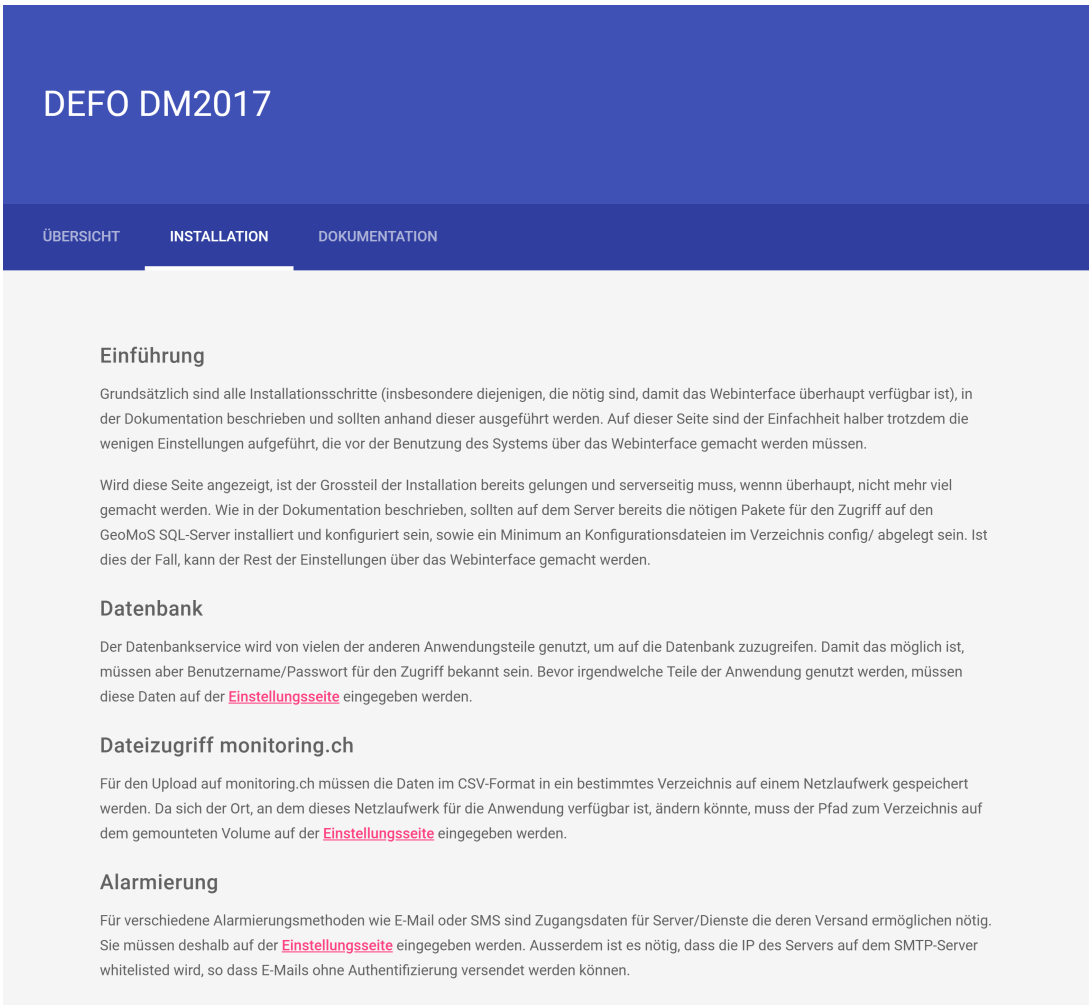


Fig. 4.19.: Installation page

After the system has been installed in a new environment, all services that require additional configuration need to be set up. Once the system runs, this can all be done inside

the UI and to make sure nothing is missed, this page has links to the configuration pages of all services that require it: database, file and notifications.

The final tab is for documentation. The first link lets the user download the PDF documentation that was written for the company and is also included in the appendix of this report. It contains technical background information about the system and a full guide for installing it from scratch in a new environment. The second link shows the Flasgger documentation of the endpoints, which is integrated into the system.

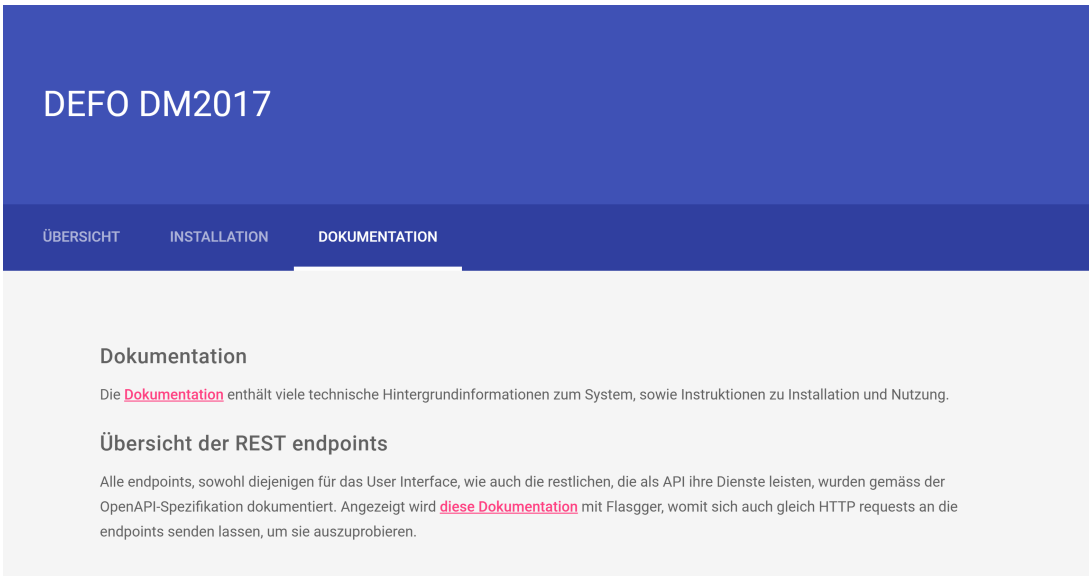


Fig. 4.20.: Documentation page

5 Scheduler

5.1. Introduction

The previous chapter has explained the different services that make up a large part of the system, but something is still missing. Users can either export measurements manually or set projects up for automated analysis and upload. The scheduler has been mentioned a few times before and it is the missing component. The different services are implemented with the REST style and as such react to requests, but have no initiative of their own. They do not "wake up" at a predetermined time to do something, but instead have to be called by a different entity. This is the scheduler in a nutshell. It is basically a separate small application running in the background, which uses only the public API provided by the services to interact with the system. It accesses project configuration to determine when users have scheduled average analysis or upload and when the time comes, uses the projects, database, notifications and file services to execute the scheduled activity.

On a technical level, the scheduler currently runs on the same server as the rest of the system (although that is not a requirement since it only interacts with the system over the service APIs) and is also started in the background on boot with a systemd unit. More on that in Appendix A about deployment. Once started, it will look at the configuration of all projects, to find out when it should do what. Given the right configuration (automation and average analysis are enabled for the project and there are times where average analysis is scheduled), it will schedule average analysis for the configured times of every project and do the same for uploads (given that again, automation is enabled and there are upload times). To facilitate the scheduling part, which is mainly keeping a queue of jobs and at any time return those that are due, a small library called schedule[1] is used. After the schedule has been prepared for the first time, the scheduler runs in an endless

[1] GitHub: `https://github.com/dbader/schedule`

loop, sleeping for a minute after every iteration. The first thing the scheduler does in an iteration is prepare a list of jobs that are due to run using the scheduler. If there are any, it will run them. This is either average analysis or export, the details of which will be discussed in the following sections. Once that is done, it will look again at all project configurations (which could have changed by now) and rebuild the schedule. This is all for one iteration and it will sleep for 60 seconds until becoming active again.

There is one more implementation detail that is worth discussing here. One can imagine that it would be very bad for the system, if the scheduler were to crash. While the system (all the services at least, which is what users interact with) would still run perfectly because of the strict separation, the automation would not work anymore. The scheduler itself is relatively simple and much care has been given to make unhandled exceptions as unlikely as possible, but it does call quite complex functions for average analysis in particular and export as well. To prevent an exception in one of these calls to crash the scheduler, Python's functools module[2] has been used to wrap these calls in an exception handling function, which will log the exception and create a notification for the user in case of a problem. This makes it impossible for an error in one of the jobs to cause a catastrophic failure of the scheduler. This measure has helped catch an ugly bug during development and theoretically improves the overall stability of the system, but it is worth noting that nothing of the sort has happened during the seven months the system has been in operation for at the time of writing.

5.2. Average Analysis

The basic idea of average analysis—as already roughly explained in the previous chapter—is to look at the most recent measurements in some time frame (e.g. the last 8 hours, depends on project configuration) for all points of a project, determine their average deviation from their original position and decide whether it is in the acceptable range or not. If it is, the most recent of the measurements is stored as the last valid measurement for the point, which will be exported during the next upload. If not, a notification for the user will be generated and no measurement is cleared for upload, because an engineer has to confirm the deviation first.

Average analysis is always done for a single project, so if two projects are scheduled for the same time, they will be processed subsequently. Average analysis for a project works as follows.

1. Get project configuration (exit with notification if there was an error).
2. Check if the project has the required configuration for average analysis (exit with notification if not).
3. For every point:
   3.1. Load point-specific settings (is it disabled, does it have value addition, ...). Skip if it is disabled.
   3.2. Get measurements for this point in the given time frame (skip point with notification if there was a problem).

[2]Documentation: `https://docs.python.org/3/library/functools.html#functools.wraps`

3.3. If there are no measurements, skip the point. In case the feature is enabled, notify the user if there were no measurements for the last n average analysis iterations.

3.4. Adjust measurements by value additions and then alarm additions if there are any.

3.5. Calculate the average of the fully adjusted measurements.

3.6. If this average deviation is larger than the threshold, build a notification. If not, remember the latest value-adjusted measurement for this point for later upload.

4. Save updated automation data (the value-adjusted latest valid measurements for all points if there were any) so it is uploaded later (notification if there was a problem).
5. Send all notifications that were created.

Notifications are created using the notifications service's API during the whole process. If one of the points has a larger than acceptable deviation, the notification will be built and sent to the `/notifications/api/add` endpoint, which will store it and return the notification's ID. This ID is kept in one of two temporary lists, depending on whether it is a "regular" notification like a point deviation or an error notification. These two lists of IDs are then handed over to the `/notifications/api/send` endpoint together with the phone numbers and email addresses of the recipients that have been set up in the project settings. There are two classes of recipients, depending on how they have been added in the project configuration (see Figure 4.5). There are those that will receive all notifications, including error messages (usually engineers), and those that will only receive "regular" ones for deviations, like customers.

5.3. Upload

Most of what was said for average analysis holds true for export as well. It is invoked in the same way by the scheduler, one project after the other. The job of upload is to go over the points and collect the latest valid measurements that were determined by an average analysis, if there are any. After that, it will build CSV data of the right format and use the file service to write a file to the appropriate location so that it is loaded by the customer website. The procedure of upload for a project is the following.

1. Get project configuration (exit with notification if there was an error).
2. Exit if there is nothing to upload.
3. Format measurements that have not yet been uploaded as CSV.
4. Send data to file service (exit with notification on error).
5. Save updated automation data to remember which measurements have been uploaded (notification if there was a problem).

Evidently, the upload process is much simpler than average analysis, because there is not a lot of logic to it. It just needs to upload the results that were cleared by a previous average analysis.

While the upload does not do anything without prior average analysis, they do not necessarily have to go together. It is perfectly possible to schedule two average analyses during

the day and then an upload in the evening, or not do uploads at all and instead use the system simply to monitor the projects and receive notifications for problems. But the standard use case would still be to have an export accompany every average analysis. To do that, the user would schedule average analysis for example for 09:00 and the export for 09:05 or even 09:00 as well. The system schedules exports after analyses, so there is no danger of having the export run before the analysis if both are planned for the same time.

5.4. Logging

By the scheduler's nature as a daemon, running in the background, it is not really transparent and therefore hard to debug. To alleviate this, it makes heavy use of Python's logging module. Most events that are of importance to the user (deviations, no measurements or other errors) already push a notification, but logging is used for more detailed information that helps in development. The logging module is quite powerful and a couple of its facilities are used. The handler is the `SysLogHandler`, which is configured to send logging messages to the local Unix syslog. The scheduler and its components also use named logger instances, so that the scheduler itself, the module for average analysis and the one for export have their own loggers. This allows fine-grained control over the logging levels of the different parts. For example, one could choose to see only messages of level *warning* or higher for average analysis, but everything (including *debug*) for the export. Needless to say, the logging levels are used extensively when creating messages. For example, deviations that are below the chosen threshold of a project are logged with level *debug*, whereas ones that surpass it (and generate a notification) are of the *warning* level. And of course, errors are created appropriately too.

Listing 5.1 has an excerpt from the syslog showing the output of average analysis for two projects. Both have the deviation threshold set to 5 mm, so everything higher than that creates a *warning* message. In this case, the loggers have been configured to show everything higher than or equal to the *debug* level. The projects contain more points than the ones shown here, the listing has been edited for brevity.

```
1  Mar 28 08:00:51 geox0019 dm-scheduler:INFO:Average analysis for project
       ZEB_Lengwil_M02
2  Mar 28 08:00:51 geox0019 dm-scheduler:DEBUG:5-537R has an average deviation of 2.1
3  Mar 28 08:00:51 geox0019 dm-scheduler:DEBUG:5-551L has an average deviation of 1.8
4  Mar 28 08:00:51 geox0019 dm-scheduler:DEBUG:5-555R has an average deviation of 3.7
5  Mar 28 08:00:52 geox0019 dm-scheduler:WARNING:5-567L has an average deviation of 5.6
6  Mar 28 08:00:52 geox0019 dm-scheduler:DEBUG:5-547L has an average deviation of 1.7
7  Mar 28 08:00:52 geox0019 dm-scheduler:DEBUG:5-567R has an average deviation of 3.7
8  Mar 28 08:00:52 geox0019 dm-scheduler:DEBUG:5-563R has an average deviation of 3.3
9  Mar 28 08:00:52 geox0019 dm-scheduler:DEBUG:5-Z504 has an average deviation of 3.2
10 Mar 28 08:00:52 geox0019 dm-scheduler:DEBUG:5-552R has an average deviation of 2.2
11 Mar 28 08:00:52 geox0019 dm-scheduler:WARNING:5-570R has an average deviation of 7.3
12 Mar 28 08:00:52 geox0019 dm-scheduler:DEBUG:5-550R has an average deviation of 2.0
13 Mar 28 08:00:52 geox0019 dm-scheduler:DEBUG:5-564L has an average deviation of 3.1
14 Mar 28 08:00:52 geox0019 dm-scheduler:WARNING:5-569R has an average deviation of 6.5
15 Mar 28 08:00:53 geox0019 dm-scheduler:DEBUG:5-533R has an average deviation of 4.0
16 Mar 28 08:00:53 geox0019 dm-scheduler:DEBUG:5-551R has an average deviation of 1.8
17 Mar 28 08:00:53 geox0019 dm-scheduler:DEBUG:5-549L has an average deviation of 1.6
18 Mar 28 08:00:53 geox0019 dm-scheduler:DEBUG:5-545R has an average deviation of 4.7
19 Mar 28 08:00:53 geox0019 dm-scheduler:DEBUG:5-550L has an average deviation of 1.7
```

```
20 Mar 28 08:00:53 geox0019 dm-scheduler:DEBUG:5-553R has an average deviation of 3.0
21 Mar 28 09:00:05 geox0019 dm-scheduler:INFO:Average analysis for project
       ZEB_Lengwil_M03
22 Mar 28 09:00:05 geox0019 dm-scheduler:WARNING:2-1016 has an average deviation of 5.3
23 Mar 28 09:00:06 geox0019 dm-scheduler:WARNING:2-505 has an average deviation of 5.5
24 Mar 28 09:00:06 geox0019 dm-scheduler:WARNING:2-134R has an average deviation of 8.6
25 Mar 28 09:00:06 geox0019 dm-scheduler:WARNING:No measurements for 2-M26_U
26 Mar 28 09:00:06 geox0019 dm-scheduler:DEBUG:2-1017 has an average deviation of 3.9
27 Mar 28 09:00:06 geox0019 dm-scheduler:DEBUG:2-319 has an average deviation of 1.1
28 Mar 28 09:00:06 geox0019 dm-scheduler:DEBUG:2-1011 has an average deviation of 4.7
29 Mar 28 09:00:06 geox0019 dm-scheduler:WARNING:No measurements for 2-123R
30 Mar 28 09:00:06 geox0019 dm-scheduler:DEBUG:2-1013 has an average deviation of 4.4
31 Mar 28 09:00:06 geox0019 dm-scheduler:DEBUG:2-315 has an average deviation of 2.6
32 Mar 28 09:00:06 geox0019 dm-scheduler:DEBUG:2-421 has an average deviation of 2.4
33 Mar 28 09:00:06 geox0019 dm-scheduler:DEBUG:2-317 has an average deviation of 3.1
34 Mar 28 09:00:07 geox0019 dm-scheduler:WARNING:2-129L has an average deviation of 5.8
35 Mar 28 09:00:07 geox0019 dm-scheduler:DEBUG:2-1014 has an average deviation of 2.8
```

List. 5.1: Excerpt from /var/log/syslog

This logging has helped tremendously during the early stages of development in particular, when it was necessary to check all individual operations to make sure the system worked correctly. The flexibility of the logging module is also excellent. By only changing a single line of code, it would be possible to have the logs sent to a socket, HTTP server or even by email. This was not necessary for this project, but it is easy to see the value in being able to have error messages sent immediately to one's phone.

6 Testing

6.1. Introduction

Unit tests and integration tests were an important part of the development workflow right from the start. With the system being relatively complex, it was clear that it had to be tested rigorously. In order not to fall behind with implementing tests, they were written simultaneously with the code they cover. This did not only help with catching bugs as they were created, it also improved the architecture of the code in question. Writing the tests required interacting with it and bad interfaces that made it hard to use became obvious very quickly this way.

6.2. Implementation

The tests were implemented with the pytest framework[1] and Python's own unit testing framework[2]. Testing the many utility functions that are part of the system is a simple affair, after all they are part of normal Python modules that can be imported and used as usual. It is a bit of a different story for the service endpoints, because they are accessed by HTTP requests. Properly testing them requires a running system that has a basic configuration, in order for database access to work. The tests then send HTTP requests to the system and analyze the results, whether this is HTML for a UI endpoint or plain JSON for an API endpoint. With the help of the requests module[3] this is very simple and does not unnecessarily litter the tests with boilerplate code. A simple test can look like the one shown in Listing 6.1. This is part of the API test for the database, which was condensed for brevity.

[1]Documentation: `https://docs.pytest.org/en/latest/`
[2]Documentation: `https://docs.python.org/3/library/unittest.html`
[3]Documentation: `http://docs.python-requests.org/en/master/`

```
import unittest
import requests
import json
from datetime import datetime, timedelta
from defo_dm2017.util import config


class DatabaseAPITestCase(unittest.TestCase):

    def setUp(self):
        kv = config.read('testconfig')
        self.db_name = kv['db_name']
        self.base_url = kv['base_url']
        self.url = self.base_url + '/database/api'
        self.p_url = self.base_url + '/database/api/points'
        self.daybefore = (datetime.now() - timedelta(days=1)).isoformat()[:19]

    def test_api_average(self):
        pdata = {
            'db_name': self.db_name,
            't_from': self.daybefore,
            'average': True
        }
        r = requests.post(self.url, data=pdata)
        rows = json.loads(r.text)
        # Test if only daily averages were returned
        for row in rows:
            self.assertTrue("23:59:59" in row[1])

    def test_api_points_500(self):
        pdata = {'db_name': 'nonexistent'}
        r = requests.post(self.p_url, data=pdata)
        resp = json.loads(r.text)
        # Check if we received the right kind of error
        self.assertEqual(r.status_code, 500)
        self.assertEqual(resp['code'], 20)

    def test_api_points(self):
        pdata = {'db_name': self.db_name}
        r = requests.post(self.p_url, data=pdata)
        response = json.loads(r.text)
        # Check if everything's fine on the server
        self.assertEqual(r.status_code, 200)
        # Check if there is some data
        self.assertTrue(len(response) > 0)
        # Check if data seems well-formed
        self.assertEqual(len(response[0]), 2)
        self.assertTrue(isinstance(response[0][0], str))
        self.assertTrue(isinstance(response[0][1], str))

    # More tests are here


if __name__ == '__main__':
unittest.main()
```

List. 6.1: Excerpt from test_database_api.py

Besides this, the database service also has separate tests for its UI and configuration endpoints. Most services and utility modules have their own tests that can be run individually. Alternatively, the whole test suite can be run.

```
(venv) geomos@geox0019:~/defo_dm2017_dev$ python setup.py pytest
running pytest
============================= test session starts ==============================
platform linux -- Python 3.5.2, pytest-3.5.0, py-1.5.3, pluggy-0.6.0
rootdir: /home/geomos/defo_dm2017_dev, inifile:
collected 86 items

tests/test_availability.py ....                                          [  4%]
tests/test_config.py ....                                                [  9%]
tests/test_database.py ...                                               [ 12%]
tests/test_database_api.py .......                                       [ 20%]
tests/test_database_config.py .                                          [ 22%]
tests/test_dbreq.py ......                                               [ 29%]
tests/test_dm_calc.py .......                                            [ 37%]
tests/test_dm_datetime.py ....                                           [ 41%]
tests/test_dm_filter.py .........                                        [ 52%]
tests/test_dm_upload.py .......                                          [ 60%]
tests/test_manual.py .......                                             [ 68%]
tests/test_notifications_api.py .......                                  [ 76%]
tests/test_projects_advanced.py ...................                      [ 98%]
tests/test_projects_basic.py .                                           [100%]

========================== 86 passed in 6.59 seconds ===========================
```

This is how it would look ideally. Running this after changes to the codebase has proven very useful, because it allows checking if no regressions were introduced.

The distinction between unit tests and integration tests is pretty fluid in this project. This is because of the dependencies between some services. For example, the projects service uses some endpoints of the database service. Even though these have their own unit tests already, the projects service as a whole is tested and by that some unit tests of the projects service also test the database service and how they work together. As such, they are also integration tests of a sort.

7 Conclusion

Because this chapter contains the author's personal experience, it is written in first person.

7.1. Review

When work on the implementation started in July 2017, there were many unknowns. At this point, I had never taken a "hands on" look at the infrastructure in place. I knew that there was a SQL server where measurements are stored, an Excel macro that is able to access this data and that in the end, CSV files had to be placed on a network share somewhere. Also, I had never worked with Flask before and only looked at the code of some basic "hello world" implementations and evaluated the general feasibility of using this web framework for our purpose. What I did know was the time budget I had allocated myself for the project. Nine weeks until the begin of the fall semester.

Despite the odds being somewhat against successful completion of the project in time, I was able to start with a good feeling because of how supportive the staff at GEOINFO was. I was very clear from the beginning about not having done something like this before and not knowing how far I would get in this limited time. I made up for this by being careful to develop in such a way that, regardless of whether I would be able to complete the whole system or not, they would still have a useful subset of it, able to function on its own and generate some value for them. The modular structure of the system played a big part in achieving this, of course. For example, even if I had only managed to implement the manual export, it still would have been useful on its own by replacing the old tool with something better. With this knowledge taking a bit of the pressure off and everybody on the same page, I had a great environment to work in.

It did take me the first week to get to know the existing infrastructure, manage to install and configure all tools to connect to the Microsoft SQL server, develop the database service to make use of this and start learning how to do things with Flask. The second week was spent mostly on learning to work with Flask as well, by improving on the database service and starting with the projects service. This "learning on the job" approach has worked well and I enjoyed being able to apply my knowledge immediately in the process. After this, time flew by and in the end I really had a fully working system with almost all of the bells and whistles that were planned and even some that were not. One thing I did not get around to doing was outlier detection using the Page-Hinckley test. This was originally intended as the method for detecting deviations, but during development

of the scheduler we opted for the simpler average analysis to save time and get a working implementation of it running as quickly as possible. In the end, it was decided to forego implementing this completely and stay with the perfectly serviceable average analysis to have enough time to realize features such as the notification sending by SMS and email, which are more useful in the big picture.

I am very satisfied with how the system turned out and it has been successfully in operation at GEOINFO since its inception. In that time it has been running in a very stable fashion, with no apparent problems and not requiring any maintenance. As far as the technology is concerned, the choices I made have held up and I would use the same stack again, with the exception of Material Design Lite. Not because it is not good, but because it has been superseded by Material Components for the Web[1], the successor of MDL. Concerning the architecture, developing a composite application with different services was absolutely worth the effort. I have already written about how this allows other applications to make full use of the system or incorporate only the functionality of parts of it. The fact that the scheduler is actually a separate application that uses the system to achieve a goal (analyze and export measurements) speaks for itself in that regard. Because the services communicate with messages sent over HTTP, it would also be possible to quite easily split the system up and have certain services run on dedicated machines in other places. To take full advantage of the increased resilience that microservices bring, one would try to have the services distributed over many machines anyway [2]. While this was not necessary for an application of this scope, there are cases where it may still be worth considering. For example, if the database was not accessible over the network, the database service could run directly on that machine. Or if some of the services required inordinate amounts of computing power (which they do not), they could be installed in environments that fulfill their demands. But the main advantage is certainly the flexibility it gives to the whole system.

7.2. Outlook

Leica Geosystems made an announcement, stating that in future versions of GeoMoS, the old direct SQL access to the database would be disabled in favor of a new API serving the same purpose. I have recently migrated the database service to use this new API instead of SQL access. Ideally, the API would provide enough functionality for the new database service to fulfill the old contract, in which case I could have simply replaced it with the new one, without having to change anything in the service consumers. Unfortunately, this is not the case and I was forced to make a few changes more than I would have liked, but it was still much less painful than such a big change could have been. Besides this, there is of course potential for testing more powerful algorithms for deviation detection or even automating more types of sensors.

[1] Website: `https://material.io/components/web/`

A Deployment

A.1. Introduction

To get a running instance of the system, some prerequisites need to be met. Some have already been mentioned, such as the database driver and driver manager that are necessary to connect to the Microsoft SQL server. There are a few more, for example small configuration files need to be prepared so the system has some essential information about its environment. All of this is already explained in detail in the documentation that was written for the company and is included in Appendix B. This chapter will only give a brief overview of the topic.

A.2. Debug

For debug and development purposes, Flask has a built-in debug server. Before starting the Flask app, some environment variables need to be set.

```
# Tell Flask which app to run
export FLASK_APP=defo_dm2017
# Enable debug (detailed error messages, reloading on file changes)
export FLASK_DEBUG=1
# Run app publicly and with threads (more than one request at a time)
flask run --host=0.0.0.0 --with-threads
```

After this, it is running and available on the network on port 5000. The host argument makes the application available "externally", meaning that other computers in the same network can access it, not only the local host. It is very important to start the debug server with the threads option, because by default it will only handle one request at a time. Since services in the system communicate with each other, this will not work. For example, the user could access an endpoint of the projects service, which in turn sends

a request to the database service. But this does not go through until the user's initial request has been handled, which will never be the case. Leaving this argument out leads to a classic deadlock situation.

In case the scheduler is also needed, it is started in a separate instance.

```
python scheduler/scheduler.py
```

All of this is very convenient during development. For one, the server gives valuable debug information in case something goes wrong, but more importantly, the server watches for file changes and reloads immediately, so it is always up to date.

A.3. Production

While very convenient, in a production environment these properties are not great. Using the debug server does not give the best performance of course, but more critically, it is very dangerous because of the inherent debugging options (= attack vectors) it provides. For these reasons, a Web Server Gateway Interface (WSGI) server is typically used. The one chosen for this project is Gunicorn[1]. This approach gives much better performance, but while it is more secure also, an application server like this is still a relatively complex piece of software and as such has considerable attack surface. Because the goal is to have the system available on port 80 and an application has to run in privileged mode to bind to it, this is a problem. If the application server were to be compromised, the attacker would have control over a privileged application on the server. Of course, this is all rather academic in this case since the system is only available inside the company network, but good practices were followed nonetheless. The solution is to use a reverse proxy. This is a very simple application (= smaller attack surface) with the sole job of forwarding requests to the actual server. Because it is less likely to be successfully attacked, it is less of a problem to have it run in privileged mode to bind to port 80.

This is the setup that has been used, with nginx[2] configured in the role of the reverse proxy. To make sure that both the Gunicorn instance serving the Flask app and the scheduler are started in case of a system reboot, systemd units are used to manage them. The nginx configuration as well as the two systemd units are in the installation directory of the project source code.

[1] Website: `http://gunicorn.org/`
[2] Website: `https://www.nginx.com/`

B
Documentation

The following pages have the documentation that was written for GEOINFO embedded in them. This documentation has all the information needed to understand the basics of the system, install it from scratch and continue developing it.

DEFO DM2017

GEOINFO Vermessungen AG

13. September 2017

Inhaltsverzeichnis

1 Einführung

Das Ziel dieses Projekts war, den Datenfluss der mit Tachymetern durchgeführten Deformationsmessungen auf das Kundenportal monitoring.ch zu automatisieren.

Die Messungen erfolgen automatisch und die Geräte werden mittels der sensor- und herstellerspezifischen Software *GeoMoS* ferngesteuert. Die Punktmessungen landen schliesslich in einer SQL-Datenbank. Bisher wurden die Daten mithilfe von VBA-Code in einer projektspezifischen Exceldatei aus der Datenbank ausgelesen und manuell bereinigt. Die aufbereiteten Daten wurden anschliessend im CSV-Format exportiert und in einem Projektverzeichnis auf einem Netzlaufwerk abgelegt, wodurch monitoring.ch Zugriff darauf erhielt.

Die Aufgabenstellung war nun, diesen Prozess zu automatisieren. So sollten die Daten richtig aufbereitet und auf Messfehler oder kritische Abweichungen überprüft werden, bevor sie auf dem Netzlaufwerk an das Portal monitoring.ch übergeben werden. Eine Alarmierungsmöglichkeit im Falle von nicht auf Messfehler zurückzuführenden Abweichungen war ebenfalls vorgesehen.

2 Technologie

Die primären Anforderungen an die Lösung waren

- Webbasiert und somit plattformunabhängig und ohne lokale Installation nutzbar
- Möglichst einfach in zukünftige Software einzubinden

Nachfolgend soll erklärt werden, inwiefern die gewählten Technologien diese Punkte erfüllen.

2.1 Python

Python hat einige Eigenschaften, welche die Sprache besonders geeignet für ein Projekt dieser Art machen. Einerseits ist Python eine *very high-level language* und hat als solche bereits viele mächtige Datentypen, wodurch sich der benötigte *boilerplate code* reduziert. Weiter ist Python eine interpretierte und keine kompilierte Sprache, was das *deployment* ein Stück weit vereinfacht. Vor allem bei der Plattformunabhängigkeit ist dies ebenfalls ein Vorteil. Nicht zuletzt wird Python mittlerweile in vielen technischen Studiengängen ebenfalls unterrichtet, wodurch es wohl die für die meisten Personen verständlichste Programmiersprache ist. Und zuletzt gibt es für Python eine Vielzahl von Frameworks, mit denen sich *RESTful webservices* implementieren lassen.

2.2 Flask

Eines dieser Frameworks ist Flask. Es ist das meistempfohlene Webframework für Projekte jeden Umfangs, wird von einer grossen Community aktiv weiterentwickelt und unterstützt Python 3. Für Flask existieren viele Erweiterungspakete und es enthält eine *template engine*, mit der sich recht einfach kleinere UI's erstellen lassen, von denen es in DEFO DM2017 einige gibt.

3 Installation

3.1 Voraussetzungen

3.1.1 Server

Das System wurde entwickelt und getestet auf einem virtuellen Server, bereitgestellt von der GEOINFO IT AG.

```
geomos@geox0019
OS: Ubuntu 16.04 xenial
Kernel: x86_64 Linux 4.4.0-83-generic
Uptime: 1d 2h 35m
Packages: 622
Shell: bash 4.3.48
CPU: 2x Intel Xeon CPU E5-2690 v3 @ 2.594GHz
RAM: 252MiB / 992MiB
```

DEFO DM2017 ist abhängig von zwei Systemen, auf die Zugriff bestehen muss.

1. Der SQL-Server von GeoMoS muss sich über das Netzwerk erreichen lassen. Dessen Name während der Entwicklung war `geov001` und testen liess sich dies somit durch `ping geov001` vom Server aus.
2. Der Netzwerkshare von dem aus monitoring.ch die verarbeiteten Daten im CSV-Format abgreift. Im Intranet unter Windows ist dies das Laufwerk `M:\`. Auf dem Server sieht es folgendermassen aus (Ausschnitt aus `df -h`)

```
Filesystem                                          Mounted on
//10.55.184.93/RGDI_DatenPOOL_Daten/GEOINFO  /data02/rgdi_datenpool_daten
```

Sowohl der SQL-Server wie auch das Netzlaufwerk müssen natürlich nicht die genau gleichen Namen haben bzw. am gleichen Ort gemountet sein, beides lässt sich später konfigurieren. In dieser Dokumentation wird aber diese ursprüngliche Konstellation verwendet, bei einem anderen Setup sind die Namen und Pfade also zu ersetzen.

3.1.2 Pip und virtualenv

Da es bei der Entwickung mit Python diverse Abhängigkeiten mit unterschiedlichen Versionen für jedes Projekt gibt, sollten diese nicht global installiert werden sondern in einer „virtuellen“ Umgebung für jedes Projekt. Deshalb muss zuerst Pip installiert werden und mit dessen Hilfe danach das Paket virtualenv.

```
sudo apt-get install python-pip
sudo pip install virtualenv
```

3.1.3 Datenbankzugriff

Abfragen von Linux an einen proprietären Microsoft SQL-Server sind leider nicht so simpel und elegant zu haben, wie das mit einem freien SQL-Server möglich wäre. Konkret werden drei Pakete benötigt.

- **FreeTDS** ist eine Sammlung von Bibliotheken die es Programmen ermöglichen, direkt mit Datenbanken auf Microsoft SQL-Server zu kommunizieren. Somit ist es ein Treiber.
- **UnixODBC** ist der Treibermanager und eine Implementation der ODBC API.
- **pyodbc** ist ein Modul für Python, das Verbindungen mittels ODBC zu Datenbanken ermöglicht.

Im Moment müssen aber nur die ersten zwei installiert werden, pyodbc ist nämlich eine Abhängigkeit des Projektpaketes und wird bei der Installation im Virtualenv automatisch installiert. Diese und die folgenden Informationen stammen aus diesem Artikel. Die benötigten Pakete können jetzt installiert werden:

```
sudo apt-get install unixodbc unixodbc-dev freetds-dev tdsodbc
```

Anschliessend kommt die Konfiguration für FreeTDS. Dazu wird am Ender der Datei

```
/etc/freetds/freetds.conf
```

eine neue Serverkonfiguration eingefügt:

```
[sqlserver]
host = geov001
port = 1433
tds version = 7.0
```

Danach kann diese Einstellung getestet werden mit dem Kommando

```
tsql -S sqlserver -U <username> -P <password>
```

Bei Erfolg kann diese Prompt mit Ctrl+D wieder verlassen werden. Um unixODBC FreeTDS verwenden zu lassen, bleibt noch

```
/etc/odbcinst.ini
```

wie folgt zu ändern:

```
[FreeTDS]
Description = TDS driver (Sybase/MS SQL)
# Some installations may differ in the paths
#Driver = /usr/lib/odbc/libtdsodbc.so
#Setup = /usr/lib/odbc/libtdsS.so
Driver = /usr/lib/x86_64-linux-gnu/odbc/libtdsodbc.so
Setup = /usr/lib/x86_64-linux-gnu/odbc/libtdsS.so
CPTimeout =
CPReuse =
FileUsage = 1
```

Zuletzt muss noch eine Datenquelle für unixODBC hinzugefügt werden. Dazu in die Datei

```
/etc/odbc.ini
```

die folgende Konfiguration einfügen:

```
[sqlserverdatasource]
Driver = FreeTDS
Description = ODBC connection via FreeTDS
Trace = No
Servername = sqlserver
```

Danach kann dies ebenfalls getester werden, indem das Kommando

```
isql -v sqlserverdatasource <username> <password>
```

ausgeführt wird. Hat dies alles geklappt, steht der Verwendung des SQL-Servers aus Python nichts mehr im Weg.

3.1.4 Initiale Konfiguration

Praktisch alle Einstellungen können über das Webinterface gemacht werden. Es gibt aber eine Datei, die für die Lauffähigkeit des Systems unverzichtbar ist und zuvor angelegt werden muss. Es handelt sich dabei um `config/production.json`. Der Ordner config wird im Projektordner angelegt, falls er noch nicht existiert. Der Inhalt der Datei ist

```
{
    "base_url": "http://172.30.22.80:5000"
}
```

Es wäre auch eine Option gewesen, dass die einzelnen Komponenten die IP-Adresse ihres Servers herauszufinden versuchen, wodurch diese Konfiguration entfallen würde. Die aktuelle Lösung ist aber flexibler für den Fall, dass alle Anfragen über einen anderen Server (Gateway/*reverse proxy*) gehen sollten, weswegen sie gewählt wurde.

3.2 Für Entwicklung

DEFO DM2017 kann mit dem in Flask eingebauten Debugserver gestartet werden. Dies hat für die Entwicklung viele Vorteile, so gibt es selbstverständlich schön aufbereitete Fehlermeldungen und der Server startet sich automatisch neu, wenn Veränderungen an den Dateien bemerkt werden. Die folgenden Befehle sind dazu im Wurzelverzeichnis des Projekts auszuführen.

```
# Create Virtualenv with python3 as python executable
virtualenv -p python3 venv
# Activate Virtualenv
. venv/bin/activate
# Install package and dependencies (flask, pyodbc) in editable mode
pip install -e .
# Tell Flask which app to run
export FLASK_APP=defo_dm2017
# Enable debug (detailed error messages, reloading on file changes)
export FLASK_DEBUG=1
```

```
# Run app publicly and with threads (more than one request at a time)
flask run --host=0.0.0.0 --with-threads
```

Danach läuft der Server und alle *endpoints* sind auf Port 5000 verfügbar, z.B.

```
172.30.22.80:5000/projects
```

Neben dem Server muss aber auch der scheduler laufen, damit die Mittelwertauswertung und der Upload zu den geplanten Zeiten stattfinden. Um den scheduler zu starten muss ebenfalls zuerst das Virtualenv aktiviert werden. Danach wird er gestartet mit

```
python scheduler/scheduler.py
```

Server und scheduler können mit Ctrl+C beendet und das Virtualenv mit `deactivate` verlassen werden. Der Ablauf zum Start des Servers ist immer der selbe, nur die Installation mit `pip install -e .` ist eine einmalige Sache und die Umgebungsvariablen FLASK_APP und FLASK_DEBUG müssen nur neu gesetzt werden, wenn sie nicht schon vorhanden sind. Sollte das System länger laufen, lohnt es sich natürlich, beide innerhalb einer Instanz von `screen` zu starten, damit sie nicht an die laufende Session gebunden sind. All die hier genannten Schritte zum Start mit dem Debugserver sind in einem Startskript enthalten. Nach einmaliger Installation (wie oben beschrieben mit `pip install -e .`), können Server und scheduler in zwei separaten Instanzen von `screen` gestartet werden mit

```
./start.sh
```

3.3 Für Produktion

Bevor mit diesem Teil begonnen wird, muss sichergestellt werden, dass das System nicht mehr läuft. Also unbedingt Server und scheduler stoppen, falls sie noch aktiv sind. Im Verlauf der Installation werden wir nginx brauchen, also muss das installiert sein.

```
sudo apt-get install nginx
```

Ausserdem fehlen noch zwei Anpassungen die mit dem veränderten Port im Vergleich zur Entwicklungsvariante zu tun haben. Zum einen muss die Datei `config/production.json` geändert werden, da nun über den Port 80 gearbeitet wird, weshalb die Portangabe `:5000` entfernt werden muss. Falls die Datei noch nicht existiert, muss sie inklusive Verzeichnis erstellt werden.

```
{
    "base_url": "http://172.30.22.80"
}
```

Schliesslich gibt es noch etwas zu verändern, damit die *endpoint documentation* auch weiterhin unter /apidocs verfügbar ist. Dazu die Datei `defo_dm2017/__init__.py` bearbeiten und im unteren Teil das *dictionary* `template` wie folgt anpassen

```
template = {
    'info': {
        'title': "DEFO␣DM2017",
        'description': "Automating␣the␣data␣transfer␣between␣GeoMoS␣"
                       "and␣monitoring.ch"
    },
    'host': '172.30.22.80'
}
```

Die Portangabe für den Port 5000 in der Eigenschaft `host` wurde entfernt, da das System mit dem Produktionsserver nun auf Port 80 erreichbar ist.

Das System wird schlussendlich so aussehen, dass wir zwei *systemd services* haben, einen für den Flask-Server und einen für den scheduler. Ausserdem wird nginx als *reverse proxy* konfiguriert, damit gunicorn, unser WSGI-Server, nicht mit erhöhten Rechten gestartet werden muss um an den Port 80 binden zu können. Die Konfigurationsdateien für diese 3 Komponenten sind im Verzeichnis `installation/` zu finden. Es lohnt sich, alle drei genau anzuschauen und möglicherweise einige Angaben zu ändern. In den .service Dateien für systemd werden absolute Pfade benutzt, also müssen sie angepasst werden, falls der Sourcecode nicht im *home directory* des Benutzers geomos abgelegt ist. Dasselbe gilt für die

Konfigurationsdatei von nginx, dabei muss möglicherweise der Pfad zum *socket* oder auch die IP-Adresse des Servers geändert werden. Sind die Dateien bereit, ist die Installation recht einfach.

```
# Copy the service file for gunicorn server to appropriate location
sudo cp installation/defoserver.service /etc/systemd/system/
# Reload systemd daemon
sudo systemctl daemon-reload
# Start the service
sudo systemctl start defoserver.service
# Make the service start on boot
sudo systemctl enable defoserver.service

# Copy nginx site file to appropriate location
sudo cp installation/defoserver /etc/nginx/sites-available/
# Create symbolic link to activate the site
sudo ln -s /etc/nginx/sites-available/defoserver /etc/nginx/sites-enabled/
# Check for syntax errors
sudo nginx -t
# Restart nginx
sudo systemctl restart nginx

# Copy the service file for the scheduler to appropriate location
sudo cp installation/defoscheduler.service /etc/systemd/system/
# Reload systemd daemon
sudo systemctl daemon-reload
# Start the service
sudo systemctl start defoscheduler.service
# Make the service start on boot
sudo systemctl enable defoscheduler.service
```

Danach sollte das System vollständig lauffähig sein, erreichbar über die IP des Servers ohne Angabe des Ports 5000. Der scheduler sollte sich ebenfalls gestartet haben, was überprüft werden kann mit `sudo cat /var/log/syslog | grep dm-scheduler`. Werden nun Änderungen am Code des Flask-Servers (in `defo_dm2017/`) vorgenommen, muss dieser manuell neu gestartet werden, damit ganz sicher die neuste Version genutzt wird. Das geht mit

```
sudo systemctl restart defoserver.service
```

Das gleiche gilt für den scheduler, also den Code im Verzeichnis `scheduler/`.

```
sudo systemctl restart defoscheduler.service
```

Soll wieder zum Setup für die Entwicklung gewechselt werden, einfach die beiden *systemd services* stoppen und deaktivieren.

```
sudo systemctl stop defoscheduler.service
sudo systemctl disable defoscheduler.service
sudo systemctl stop defoserver.service
sudo systemctl disable defoserver.service
```

Dazu noch die wegen dem Port veränderten Dateien wieder in ihren Ursprungszustand mit :5000 an der IP Adresse zurücksetzen. Dann wie gehabt den Debugserver starten und weiterentwickeln.

3.4 Konfiguration über das Webinterface

Ist das System erst einmal installiert, müssen für den Betrieb noch einige letzte Einstellungen gemacht werden. Sie sind in der Sektion „Installation“ auf der Übersichtsseite des Systems aufgeführt, inklusive Links zu den Orten an denen sie gemacht werden können. Die Übersichtsseite ist verfügbar auf dem relativen Pfad /, also z.B. 172.30.22.80:5000 mit dem Debugserver, oder nur die IP ohne die Portangabe auf dem Produktionsserver. Die Einstellungen beinhalten unter anderem die Zugangsdaten für den SQL-Server, den Pfad zum Export der CSV-Dateien oder Logins für die Benachrichtigung per E-Mail und SMS. Die Einstellungen sind also unverzichtbar um alle Funktionalitäten nutzen zu können und sollten sofort gemacht werden, bevor mit der Nutzung des Systems begonnen wird.

3.5 Backup

Alle im Webinterface gemachten Einstellungen und die restlichen Daten, die während dem Betrieb anfallen, werden im Verzeichnis `config/` im JSON-Format abgelegt. Das soll die Wiederherstellung des Systems im Fall einer Migration vereinfachen. Um diese Daten regelmässig zu sichern, kann irgend ein passendes Werkzeug verwendet werden. Hier wird beschrieben, wie die Lösung auf dem System, auf dem DEFO DM2017 entwickelt wurde, aussah.

Nebst dem für monitoring.ch nötigen Netzlaufwerk, das auf `/data02/` gemountet wurde, war auch eine Partition des GeoMoS-Servers unter `/data01/` vorhanden. Diese wurde genutzt, um mithilfe von Rsync und cron eine tägliche Kopie der Konfiguration abzulegen. Dazu wurde mit dem Befehl `crontab -e` die *cron table* für den aktuellen Benutzer geöffnet und die Zeile

```
0 21 * * * rsync -a /home/geomos/defo_dm2017/config/ /data01/rgdi_
datenpool_daten/$(date +\%F)
```

eingefügt. Aus Platzgründen wurde in diesem Dokument ein Zeilenumbruch benutzt, tatsächlich ist es nur eine zusammenhängende Zeile. Damit wird der Inhalt des Konfigurationsverzeichnisses jeden Tag um 21:00 Uhr auf die erwähnte Partition des GeoMoS-Servers in einen Ordner mit dem aktuellen Datum als Namen kopiert. Zu beachten ist dabei, dass im Teil `$(date +\%F)` das Prozentzeichen *escaped* ist, da dies in cron nötig ist.

4 Allgemeine Hinweise

4.1 Sicherheit

Das System wurde für den Einsatz im gesicherten lokalen Netzwerk, also nicht aus dem Internet zugänglich konzipiert und umgesetzt. Als solches ist es für alle Teilnehmer im LAN erreichbar und auch nicht durch Benutzername/Passwort geschützt. Da den Angestellten grundsätzlich Vertrauen entgegengebracht wird, wurde Zeit in die Entwicklung von Features anstatt Schutz vor allen möglichen Attacken investiert. Natürlich wird verhindert, dass Nutzer aus Versehen Schaden anrichten (so werden z.B. Inputs auf ihre grobe Richtigkeit überprüft), aber das System wurde ausdrücklich nicht dafür entworfen, sich gegen bösartige Angriffe zu schützen.

4.2 Browserkompatibilität

Bei der Entwicklung wurde darauf geachtet, den Stack möglichst klein zu halten um die technische Komplexität zu reduzieren. So wird zum Beispiel nur an ganz wenigen ausgewählten Stellen überhaupt clientseitiges JavaScript ausgeführt und auch dort nur in einem Rahmen, der keine zusätzlichen Bibliotheken wie z.B. jQuery nötig machen würde. Alle anderen Aufgaben werden über *HTTP requests* an den Server übermittelt und dort erledigt. Auch für das User Interface wird auf im Browser bereits vorhandene Funktionalität zurückgegriffen. Für einigermassen ansehliche Darstellung wird Material Design Lite verwendet, eine schlanke Bibliothek, die hauptsächlich durch modernes CSS die üblichen HTML-Steuerelemente etwas auffrischt.

Der geübte Leser wird bereits ahnen, dass an dieser Stelle gleich auf die Unvereinbarkeit des Wortes „modern“ mit beinahe allen Versionen von Internet Explorer hingewiesen wird. Und tatsächlich sieht das mit diesen Technologien entwickelte System auf IE 11 nicht nur alles andere als schön aus, sondern ist praktisch unbenutzbar weil neuere Webstandards von diesem Browser nicht unterstützt werden. Auf Kompatibilität mit Internet Explorer wurde kein Wert gelegt, denn diese Zeit wäre wirklich am falschen Ort investiert gewesen. Entwickelt und getestet wurde DEFO DM2017 mit Mozilla Firefox, aber die Nutzung sollte mit jedem *evergreen browser* ohne Probleme möglich sein.

4.3 Zeitangaben für die Automatisierung

In den Projekteinstellungen können für den Automatikmodus Aufgaben wie die Mittelwertauswertung oder der Upload zu gegebenen Zeitpunkten eingeplant werden. Dafür sind zwei Textfelder verfügbar, in denen Wochentage und Uhrzeiten eingegeben werden können. Die Aufgaben werden dann mit einer Genauigkeit von etwa einer Minute zum angegebenen Zeitpunkt ausgeführt. Es ist möglich, im gleichen Projekt oder sogar über mehrere Projekte hinweg, Aufgaben zum gleichen Zeitpunkt auzuführen. Das führt zu keinen Konflikten, da tatsächlich nichts gleichzeitig geschieht, sondern die Aufgaben sequenziell,

eine nach der anderen ausgeführt werden. Die Reihenfolge in der dies geschieht ist dann aber nicht vorgegeben. Deshalb ist es eine schlechte Idee, für ein Projekt die Mittelwertauswertung und den Upload gleichzeitig laufen zu lassen, da der Upload möglicherweise vorher gestartet wird. In solchen Fällen wird empfohlen, den Upload frühestens eine Minute nach der Mittelwertauswertung einzuplanen.

Der scheduler liest jede Minute die Projektkonfiguration neu, um auf allfällige Planänderungen zu reagieren. Um Probleme zu vermeiden, verzichtet er aber darauf, wenn weniger als 2 Minuten bis zu einer bereits geplanten Aufgabe verbleiben. Deshalb müssen neue Aufgaben um sicher zu gehen mindestens 3 Minuten vor ihrer geplanten Ausführung eingegeben werden, da sie sonst möglicherweise nicht rechtzeitig eingelesen werden und die Ausführung erst eine Woche später stattfindet.

Wenn weniger als zwei Minuten nach einer Aufgabe (aber nicht zum gleichen Zeitpunkt) eine weitere geplant ist, wird der scheduler warten und beide Aufgaben zum späteren Zeitpunkt ausführen. In diesem Fall werden sie aber in der geplanten Reihenfolge ausgeführt.

4.4 Punkteinstellungen

Es gibt gewisse Einstellungen, die sowohl für das Projekt wie auch für einzelne Punkte gemacht werden können. Für Punkte ohne eigene Einstellungen gelten die Projekteinstellungen. Sobald aber eine Einstellung für einen Punkt gemacht wird, hat er von da an seine eigenen. Zu dem Zeitpunkt werden (abgesehen von der veränderten Einstellung) die restlichen Einstellungen vom Projekt übernommen und als Punkteinstellungen gespeichert. Das bedeutet, dass sich Veränderungen an den Projekteinstellungen nun nicht mehr auf diesen Punkt auswirken, da er eigene hat. Das darf nicht vergessen werden, denn wird für einen Punkt zum Beispiel eine Werteaddition vorgenommen, hat der Punkt von da an seine eigenen Einstellungen. Wird danach z.B. auf Projektebene die Alarmierungsschranke verändert, wirkt sich das nicht auf diesen Punkt aus, der immer noch mit der Alarmierungsschranke von damals arbeitet, als er seine eigenen Einstellungen bekommen hat.

Wenn von der Benachrichtigung einer Abweichung aus eine Messung zum Upload freigegeben oder eine Werteaddition vorgenommen wird, gibt es die Option, die Alarmierungsschranke für diesen Punkt zu ändern. Dabei wird der bisher verwendete Wert vorgeladen, also entweder derjenige der Projekteinstellungen wenn der Punkt vorher noch keine eigenen hatte, oder der Wert der Punkteinstellungen andernfalls. Wird dieser Wert nicht verändert oder ganz weggelassen, werden keine spezifischen Einstellungen für diesen Punkt gemacht, da sich nichts verändert hat. Ein Punkt ohne eigene Einstellungen wird somit weiterhin diejenigen des Projekts verwenden, wenn der Wert nicht verändert wird. Wird aber eine neue Alarmierungsschranke für den Punkt gewählt, gilt es wieder aufzupassen. Von da an hat er nämlich eigene Einstellungen und Änderungen der Projekteinstellungen haben keinen Effekt mehr auf ihn.

4.5 Projekteinstellungen für Automatisierung und Alarmierung

Dieser Abschnitt steht im Zusammenhang mit dem vorhergehenden und soll ein mögliches Missverständnis aufklären. Gehen wir von einem Szenario aus, in dem für einen Punkt eine Einstellung gemacht wird (z.B. eine Werteaddition), während der Automatikmodus für das Projekt aktiv ist. Er übernimmt also die Projekteinstellungen als Punkteinstellungen und abgesehen von der Werteaddition verhält er sich genau wie vorher. Anhand der Erklärung im vorhergehenden Abschnitt könnte man jetzt vermuten, dass wenn der Automatikmodus für das Projekt deaktiviert wird, sich das nicht auf den Punkt auswirkt, da er in seiner Einstellung den Automatikmodus aktiviert hat. Das wäre aber ungünstig, schliesslich möchte man mit dem projektweiten Schalter den Automatikmodus für das ganze Projekt abschalten, und das nicht noch manuell für alle Punkte machen müssen, die eigene Einstellungen haben.

Tatsächlich verhält sich das System in diesem Fall aber so, wie es sollte. Ist der Automatikmodus für ein Projekt deaktiviert, wird die Mittelwertauswertung gar nicht erst gestartet, weshalb die individuellen Punkteinstellungen dann nicht mehr relevant sind. Auch die projektweite Einstellung für die Alarmierung verhält sich ähnlich. Ist die Alarmierung im Projekt deaktiviert, wird unabhängig von den Punkteinstellungen nicht alarmiert. Die umgekehrte Variante für beide Fälle, also dass Automatik oder Alarmierung für das Projekt aktiviert sind aber für einzelne Punkte deaktiviert, funktioniert selbstverständlich ebenfalls wie erwartet. Und ist in einem Projekt (oder Punkt) die Automatik nicht aktiviert, findet für dieses Projekt (oder den Punkt) natürlich keine Alarmierung statt, da dafür eine Mittelwertauswertung nötig wäre.

4.6 Fehlermeldungen und Alarmierung

Die meisten Benachrichtigungen werden von der Mittelwertauswertung generiert. Davon gehört der grösste Teil zu den „regulären“ Benachrichtigungen, die über eine Abweichung oder fehlende Messungen eines Punktes informieren. Treten während der Analyse oder des Uploads aber Systemfehler auf, werden diese auch angezeigt. Da möglicherweise gewisse Empfänger eines Projekts nur an Updates zu den Punkten, nicht aber an Fehlern interessiert sind, gibt es dabei eine Aufteilung. Wird ein Empfänger (sowohl bei E-Mail wie auch SMS) normal in den Projekteinstellungen eingetragen, erhält diese Person nur während der Mittelwertauswertung generierte Benachrichtigungen, die sich entweder auf Abweichungen oder fehlende Messungen beziehen. Wird aber ein Stern '*' vor den Empfänger gesetzt (ob mit oder ohne Abstand ist unwichtig), erhält diese Person nicht nur die „regulären“ Benachrichtigungen, sondern auch Fehlermeldungen des Systems.

4.7 SMS-Versand

Für den Versand von SMS wurde ein Konto bei der iNetWorx AG eröffnet. Gewählt wurde dieser Anbieter wegen der schnellen und unkomplizierten Beratung und dem vorteilhaften Angebot. Es gibt keine monatliche Grundgebühr und kein unnötig kompliziertes Verrechnungssystem mit Punkten, von denen eine SMS eine bestimmte Anzahl kostet, abhängig vom Provider des Empfängers. Stattdessen wird im Voraus eine bestimmte Anzahl SMS gekauft und das ist alles. Je nach eingekaufter Menge variiert der Preis, mit dem Paket über 5000 SMS bezahlt man 11.2 Rappen pro SMS, bei 10000 Stück bereits nur 9.3 Rappen. Angeboten wird auch ein Web-Client, über den manuell SMS versendet werden können. Ist der Kontostand niedrig, wird der Kunde rechtzeitig kontaktiert, um ein weiteres Paket zu kaufen. Zur Sicherheit lässt sich in DEFO DM2017 auch einstellen, ab wie vielen verbleibenden SMS gewarnt werden soll. Nach jedem Versand wird dabei über das API geprüft, ob noch eine genügende Anzahl verbleibt, falls nicht wird im Benachrichtigungscenter eine Warnung angezeigt.

5 Entwicklung

5.1 Design

Das System wurde möglichst modular als Sammlung einzelner Services entwickelt, die API's anbieten und bei Bedarf darüber miteinander kommunizieren. Somit können einzelne Services problemlos in andere Anwendungen eingebunden und über *HTTP requests* genutzt werden. Da die primäre Aufgabe war, eine Lösung für die Verarbeitung von Tachymeterdaten zu entwickeln, ist der grösste Teil des Systems natürlich darauf ausgerichtet und nicht einfach so für andere Zwecke nutzbar. Zum Beispiel sind die ganze Auswertung und der Export auf das Datenformat der Tachymetermessungen angewiesen, schliesslich ist deren Verarbeitung ihre Aufgabe. Wo immer es aber möglich war, wie bei `notifications` oder `file`, wurden komplett unabhängige Services entwickelt, die ohne Anpassungen auch für andere Zwecke einsetzbar sind. Umgesetzt wurden die einzelnen Services als *Flask blueprints*, es kann also auch nur eine Untermenge der Services gestartet werden, indem nur die gewünschten in `defo_dm2017/__init__.py` registriert werden. Eine kurze Übersicht über die Services findet sich im Kapitel 5.5 zu den Komponenten.

5.2 i18n

Mehrsprachigkeit war kein Fokus bei der Konzeption und Entwicklung von DEFO DM2017. Das System löst ein spezifisches Problem für die GEOINFO Vermessungen AG und es gibt keine Pläne für einen Einsatz in einer anderen Sprachregion. Da der nötige Zusatzaufwand für einen Grundaufbau, der mehrere Sprachen ermöglichen würde den Ertrag weit übersteigt, wurde im Interesse der Zielerfüllung darauf verzichtet.

5.3 Tests

Im Ordner `tests/` sind alle Tests für die diversen Komponenten des Systems abgelegt. Getestet werden sowohl die direkt aus dem Code verwendeten Module wie config.py oder dbreq.py, wie auch die *Flask endpoints* die im Browser bzw. über *HTTP requests* aufgerufen werden. Die Testskripts für letztere verwenden ebendiese Art um sie aufzurufen, was zwar der realen Anwendung entspricht, es aber praktisch unmöglich macht, die tatsächliche *code coverage* festzustellen. Man kann zwar mithilfe von `coverage`

die Tests laufen lassen, wird aber feststellen, dass in den views.py (Flask) nur die Zeilen mit den Methodensignaturen aufgerufen werden, da der Flask Server in einem separaten Prozess läuft und somit nicht überwacht werden kann. Auch wenn es theoretisch möglich wäre, wurde keine Zeit investiert um dieses kosmetische Manko zu beheben. Wichtig ist, dass die tatsächliche *code coverage* grösser ist als die gemessene.

Da die Tests also gewisse Teile des Systems per *HTTP requests* nutzen, muss folglich der Server gestartet sein damit sie erfolgreich sind. Ebenfalls wichtig ist, dass das System bereits grundsätzlich konfiguriert wurde (Username/Passwort in /database/config gesetzt), da dies Voraussetzungen für den Datenbankzugriff sind. Weiter werden zusätzliche Informationen von den Tests benötigt, nämlich der Name einer Datenbank (mit Punktmessungen darin) um damit den Zugriff zu testen, sowie die URL (mit Port, falls nicht 80) über die die Services verfügbar sind. Diese Informationen werden von den Tests mit dem Modul config.py gelesen und müssen deshalb in der Datei `config/testconfig.json` abgelegt werden. Beigelegt ist eine Testkonfiguration zu sehen, werden die Tests auf dem Produktionsserver ausgeführt muss die Portangabe entfernt werden, da dann alles über 80 laufen wird.

```
{
    "db_name": "Uster",
    "base_url": "http://172.30.22.80:5000"
}
```

Zusammengefasst sind also die Voraussetzungen für erfolgreiche Tests:

1. Server mit System läuft
2. Benutzername und Passwort für Datenbankzugriff sind eingestellt (/database/config)
3. Datei `config/testconfig.json` vorhanden und korrekt (inkl. Portangabe oder nicht, abhängig davon ob Debug- oder Produktionsserver im Einsatz ist)

Sind alle Bedingungen erfüllt, können die Tests gestartet werden. Dazu innerhalb des Wurzelverzeichnisses die folgenden Befehle ausführen.

```
. venv/bin/activate
python setup.py pytest
```

Die für die Tests benötigten Abhängigkeiten sind in setup.py deklariert und werden so automatisch installiert. Deshalb ist es wichtig, dass zuerst das Virtualenv aktiviert wird.

5.4 Dokumentation

Nebst dieser Dokumentation wurde der Code selbst auch dokumentiert. Für „normale“ Pythonmodule die direkt aus Code aufgerufen werden (z.B. dbreq.py oder config.py) wurden *docstrings* geschrieben, die der NumPy-Konvention entsprechen. Diese unterscheidet sich gegenüber normalen *docstrings* insofern, als dass sie Vorgaben darüber macht, wie Parameter und Rückgabewerte dokumentiert werden. Zusätzlich wurden die *Flask endpoints* mit Flasgger dokumentiert. Flasgger ist eine Erweiterung für Flask, die der OpenAPI-Spezifikation entsprechende Dokumentation aus den *docstrings* der *Flask views* auf einer Webseite mithilfe von Swagger UI schön darstellt. Dies ist ein sehr nützliches Tool, da an einem Ort alle Teile des Systems ersichtlich sind. Die *endpoints* wurden dabei nach Services zu denen sie gehören geordnet. Die meisten sind lediglich für das User Interface zuständig und akzeptieren somit typischerweise irgendwelche Parameter als *POST request* aus einem HTML-Formular und geben eine Seite in HTML zurück, aber es gibt auch welche, die eine reine API-Funktion haben (z.B. /database/api oder /projects/api) und von anderen Services und dem User Interface genutzt werden. Diese *endpoints* lassen sich auch problemlos von anderen Anwendungen aus verwenden. Die hier beschriebene Übersicht lässt sich aufrufen unter /apidocs.

Damit diese (PDF-)Dokumentation nicht verloren geht und immer verfügbar ist für Nutzer von DEFO DM2017, ist auf der Übersichtsseite im Tab „Dokumentation“ zusammen mit einem Link auf die Flasgger-Seite auch ein Link auf dieses Dokument aufgeführt. Konkret wird dort einfach auf `/defo_dm2017/static/Dokumentation.pdf` verlinkt. Es ist also wünschenswert, dass diese Datei bei Änderungen an diesem Dokument ebenfalls aktualisiert wird, damit immer nur eine Version im Umlauf ist.

5.5 Komponenten des Systems

In diesem Abschnitt sind alle Komponenten aufgeführt, mit einer kurzen Erklärung ihrer Funktion und den Abhängigkeiten zu anderen Services.

5.5.1 autocontrol

Dient dazu, den automatischen Upload etwas transparenter zu machen. Dabei werden für ein Projekt alle aktiven Punkte angezeigt, mit der letzten gültigen Messung, die beim nächsten Upload exportiert würde. Ausserdem gibt es die Möglichkeit, den Upload manuell vorzuziehen.

Benötigt	Grund
projects	Einstellungen abfragen und aktualisieren
file	Export für monitoring.ch
overview	Link zurück auf Übersicht

5.5.2 database

Wird von anderen Komponenten genutzt, um Abfragen an die Datenbank zu machen. Muss zuerst über die eigene Einstellungsseite mit Benutzername/Passwort für die GeoMoS-Datenbank konfiguriert werden.

5.5.3 decision

Ermöglicht von einer Benachrichtigung wegen einer Abweichung aus entweder eine Werteaddition vorzunehmen und freizugeben, oder nur eine Messung zum Upload freizugeben.

Benötigt	Grund
projects	Einstellungen abfragen und aktualisieren
notifications	Löschung der Benachrichtigung, über die der Nutzer herkam

5.5.4 file

Bietet Dienst zum schreiben von Dateien an einen vorher konfigurierten Ort. Dieser wird über die eigene Einstellungsseite ausgewählt.

5.5.5 manual

Der manuelle Export der neusten Messungen eines Projekts.

Benötigt	Grund
projects	Einstellungen abfragen und aktualisieren
database	Abfrage von Messungen
file	Export für monitoring.ch
overview	Link zurück auf Übersicht

5.5.6 notifications

Das Benachrichtigungscenter für Übersicht und Bearbeitung von Benachrichtigungen, sowie Versand per SMS und E-Mail. Dazu müssen Zugangsdaten auf der eigenen Einstellungsseite eingegeben werden. Für den Versand per E-Mail ist es zudem nötig, dass auf dem angegebenen SMTP-Server die IP unseres Servers whitelisted wird, so dass das System ohne Authentifizierung E-Mails mit dem angegebenen Absender versenden kann.

5.5.7 overview

Übersichtsseite mit Links zu allen Teilen, sowie Installationsinstruktionen für Einstellungen, die über das Webinterface gemacht werden können und Dokumentation.

Benötigt	Grund
notifications	Link auf Benachrichtigungscenter und Einstellungen
projects	Abfrage von Projektnamen für Links
manual	Link auf manuellen Export für Projekte
autocontrol	Link auf Kontrolle automatischer Export für Projekte
database	Link auf Einstellungen Datenbank
file	Link auf Einstellungen Dateiexport

5.5.8 projects

Projektübersicht, in der Projekte erstellt und bearbeitet werden können.

Benötigt	Grund
database	Abfrage von Punkten in Datenbank
overview	Link zurück auf Übersicht

5.5.9 scheduler und periodische Aufgaben

Um zu gewährleisten, dass diverse Aufgaben wie z.B. die Mittelwertanalyse oder der Upload der Daten auf monitoring.ch zu den vom Benutzer ausgewählten Zeitpunkten geschehen, ist eine Art Daemon nötig, der die ganze Zeit läuft und sie zur gewünschten Zeit startet. Es wäre eine Möglichkeit gewesen, dafür bereits existierende Tools wie cron zu verwenden, aber eine optimale Lösung sieht anders aus. Bei der Entwicklung hätte sich zwar vielleicht Zeit sparen lassen, die aber in Installations- und Konfigurationsaufwand geflossen wäre. Neue Abhängigkeiten zu Systemtools wären entstanden und die Einbindung, gerade von cron, hätte sich auch schwierig gestaltet. Schliesslich wäre es nötig gewesen, bei Änderungen der Zeitpunkte die *crontable* ebenfalls aktuell zu halten und unter dem Strich wäre das eine Bastelei gewesen.

Deshalb wurde entschieden, eine Lösung ebenfalls in Python zu implementieren. Leider kann sie nicht in den Flask-Server integriert werden, da Flask typischerweise auf einem WSGI-Server mit mehreren Threads läuft. Das hätte zur Konsequenz, dass unser scheduler (ohne aufwändige *workarounds*) ebenfalls in mehreren Instanzen ausgeführt würde und das taugt nichts. Somit muss der scheduler in einem separaten Prozess unabhängig von Flask laufen. Der scheduler, sowie die Module die Funktionalität wie die Mittelwertauswertung oder den Upload enthalten, befinden sich im Unterverzeichnis `scheduler/`. Der scheduler greift jede Minute über den /projects/api *endpoint* auf die Projekteinstellungen zu und prüft, ob es an der Zeit wäre, Jobs auszuführen. Ist dies der Fall, werden die entsprechenden Funktionen aufgerufen. Da der scheduler im Hintergrund arbeitet und die Vorgehensweise nicht sofort ersichtlich ist, führt er relativ detaillierte Logs. Die Anzahl an Informationen die aufgezeichnet werden lässt sich über den *log level* im Sourcecode verändern. Konfiguriert ist der Logger so, dass die logs ins syslog aufgenommen werden, das unter `/var/log/syslog` verfügbar ist. Der Vorteil dieser Methode (gegenüber einer eigenen Logdatei z.B. im Anwendungsverzeichnis) ist, dass dieses Log von `logrotate` behandelt wird. Dieses ist standardmässig so konfiguriert, dass ältere Logs zuerst komprimiert und nach längerer Zeit gelöscht werden, um ständig wachsenden Speicherverbrauch zu verhindern. Da aber alle möglichen Anwendungen ins syslog protokollieren, kann die Ausgabe „gefiltert“ werden, um nur Informationen zum scheduler und den davon benutzten Modulen zu sehen.

```
sudo cat /var/log/syslog | grep dm-scheduler | less
```

Um mehrere Log-Dateien anzuschauen (auch solche, die bereits komprimiert sind), kann `zless` verwendet werden.

```
sudo zless /var/log/syslog /var/log/syslog.1 /var/log/syslog.2.gz
```

5.6 Projekt- und Punkteinstellungen

Einer der komplexesten Teile des Systems ist zweifelsohne die Projekteinstellung. Zum einen gibt es die für das Projekt gültigen Einstellungen, diese werden durch Klick auf „Speichern“ per *POST request* an den Server gesendet und dort verarbeitet. Soweit so einfach. Wirft man einen Blick auf die Funktion `projects_name_settings_post()` in `defo_dm2017/projects/views.py` ist dort zu sehen, dass dieser *endpoint* alle im User Interface anpassbaren Projekteinstellungen entgegennimmt. Es werden drei Typen von Parametern unterschieden, da mit ihnen unterschiedlich umgegangen werden muss.

- `numbers` sind alle Zahleneingaben
- `strings` sind Texteingaben
- `switches` sind Schalter

Wird also eine neue Projekteigenschaft im User Interface, das heisst in der Datei `defo_dm2017/projects/templates/project.html` hinzugefügt, muss der Inhalt des entsprechenden Tupels angepasst werden. Es gibt aber gewisse Textfelder, deren Eingabe zwingend ist, sofern ein bestimmter Schalter aktiviert ist. Der folgende Abschnitt ist nur in diesen Fällen anwendbar.

Damit das dem Benutzer auch visuell gezeigt wird, sobald er den Schalter betätigt, muss clientseitig etwas JavaScript ausgeführt werden. Damit dieser Code aber weiss, welche Eingaben mit welchem Schalter zusammenhängen, müssen sie dort ebenfalls aufgelistet sein. Zuerst muss in der Funktion `prepareSwitches()` eine Funktion für den Switch hinzugefügt werden, sollte noch keine existieren. Dann muss in dieser Funktion die Funktion `makeRequired()` mit der ID des Textfeldes aufgerufen werden. Somit wird für das Textfeld die Eigenschaft `required` im DOM gesetzt. Damit das visuell aber angezeigt wird, muss das entsprechende Textfeld per Script „refreshed“ werden. Das wird von der Funktion `refreshInputs()` erledigt, die dies für alle Felder in der globalen Variable `pr_numbers` erledigt. Das entsprechende Feld muss also schlussendlich auch dort noch eingefügt werden. Die Variable hat zwar „numbers“ im Namen, gilt aber für alle Textfelder, auch für solche die in `views.py` bei den `strings` aufgeführt sind.

Weiter gibt es noch die Punkteinstellungen, deren Bearbeitung dynamisch erfolgt. Konkret werden diese Änderungen per Ajax in einem *POST request* asynchron an den Server übermittelt. Hier geschieht die Arbeit der Auslese der Einstellungen bereits clientseitig, gesendet werden sie anschliessend als JSON an die Funktion `projects_api_name_pointupdate_post()` in `views.py`. Diese speichert die Einstellungen schliesslich nur noch, ist also wesentlich simpler und weiss nicht, um welche Eigenschaften es sich genau handelt. Das heisst aber, dass der clientseitige Code hier den Überblick über die Eingabefelder haben muss. Konkret müssen also alle Switches der Punkteinstellungen in der globalen Variable `po_switches` und die Zahlen (in den Punkteinstellungen gibt es nur numerische Eigenschaften, keine *strings*) in `po_numbers` aufgeführt sein.

Auch hier gilt für Textfelder, die zwingend sind abhängig von einem Schalter, dass in der Funktion für den Schalter mithilfe der ID des Textfeldes dessen Zustand aktualisiert wird. Häufig gibt es Einstellungen, die in den Projekt- wie den Punkteigenschaften vorhanden sind. In diesen Fällen wird für die ID der Punkteinstellung einfach „p_“ als Präfix verwendet wenn das `<input>` Feld in HTML definiert wird. Die Funktion `prepareSwitches()` weiss durch einen Parameter, ob es sich um eine Projekt- oder Punkteigenschaft handelt, und fügt der ID nach Bedarf den Präfix hinzu.

Schlussendlich gibt es noch einen Ort, an dem aktualisierte Punkteinstellungen nachgeführt werden müssen. Es gibt nämlich den *endpoint* für Entscheidungen, in dem der Benutzer über die Benachrichtigung einer Abweichung entweder eine Werteaddition vornehmen oder eine Messung zum Upload freigeben kann. Wird ersteres gemacht, wird also eine Werteaddition in den Punkteinstellungen vorgenommen. Sind aber noch keine Einstellungen für diesen einen Punkt vorhanden, müssen sie ausgehend von den Projekteinstellungen übernommen werden. Die Funktion `prepare_point()` in `defo_dm2017/decision/util.py` hat deshalb ebenfalls zwei Tupel, einen für die Schalter und einen für die Zahlen. Bei Änderungen in den Punkteinstellungen müssen diese also auch aktualisiert werden.

Die ganze Lösung ist leider nicht besonders flexibel oder übersichtlich, sie ist wegen gewachsenen Anforderungen so entstanden. Hier besteht sicher noch Verbesserungspotenzial, bei Änderungen ist aber wichtig zu beachten, dass die Art, wie die Einstellungen schlussendlich in der Konfigurationsdatei sind, gleich bleibt, da die restlichen Systemteile darauf ausgelegt sind.

5.7 Fehler

Die meisten Komponenten des Systems haben die Möglichkeit, Fehler zurückzugeben, falls welche auftreten sollten. Die *endpoints* die als UI fungieren werden die Fehler typischerweise als HTML anzeigen und dem Browser durch den *HTTP status code* das Problem signalisieren. Die API-*endpoints* werden das Problem ebenfalls durch den *HTTP status code* andeuten, aber zusätzlich als Antwort ein JSON-*dictionary* senden, das die Schlüssel `code` und `message` beinhaltet. Der Code ist eine ganze Zahl und die Nachricht die dazu gehörende Fehlermeldung. Da gewisse dieser Fehler an mehreren Stellen im Code produziert werden können und es deshalb nötig ist, die Übersicht darüber zu behalten, um Duplikate zu vermeiden, ist hier eine Auflistung aller vom System produzierten Fehler.

code	**message**
10	Kein Benutzername/Passwort für Datenbank vorhanden
20	Fehler bei Datenbankzugriff. Möglicherweise ist die angegebene Datenbank nicht vorhanden oder Benutzername/Passwort sind nicht korrekt
30	Dieses Projekt existiert nicht.
40	Keine Daten für diesen Punkt im angegebenen Zeitraum.
50	Mindestens eine Variable (E/N/H/2D/3D) muss zur Überprüfung ausgewählt sein.
60	Dieser Punkt wurde nicht für das Projekt aktiviert oder existiert gar nicht.
70	Die Benachrichtigung entspricht nicht dem vorgegebenen Format.
80	ID wird für Löschung benötigt.
90	ID der Benachrichtigung ist keine Zahl.
100	ID der Benachrichtigung existiert bereits.
110	Ungültiges JSON erhalten.
120	Konnte Projekte nicht abfragen.
130	Dateiexport ist nicht konfiguriert, kein Zielpfad vorhanden.
140	Fehler bei Dateizugriff. Vielleicht existiert das Verzeichnis nicht.

C
Common Acronyms

AJAX	Asynchronous JavaScript And XML
API	Application Programming Interface
CIFS	Common Internet File System
CSS	Cascading Style Sheet
CSV	Comma Separated Values
DOM	Document Object Model
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ID	Identifier
IP	Internet Protocol
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
LAN	Local Area Network
MDL	Material Design Lite
ODBC	Open Database Connectivity
REST	Representational State Transfer
SMTP	Simple Mail Transfer Protocol
SOA	Service Oriented Architecture
SQL	Structured Query Language
UI	User Interface
URI	Uniform Resource Identifier
VBA	Visual Basic for Applications
WSGI	Web Server Gateway Interface

D
License of the Documentation

E
CD-Rom

Overview of the content

```
/
├── src/
│   ├── defo_dm2017/
│   ├── documentation/
│   ├── installation/
│   ├── scheduler/
│   └── tests/
└── swagger-ui/
```

Short description

defo_dm2017

Flask app code, which consists of the different services (blueprints) and utility functions.

documentation

The full documentation of the system, which was written for the company.

installation

The files for installation. These are two systemd units for the Flask app and scheduler respectively, as well as a small nginx configuration for the server.

scheduler

Code for the scheduler: the scheduler itself, average analysis, upload and utilities.

tests

All test cases.

swagger-ui

A local Swagger UI distribution with the structure of the Flask app embedded. Open index.html in a web browser to view the documentation of services and endpoints.

References

[1] Thomas Erl, Benjamin Carlyle, Cesare Pautasso, and Raj Balasubramanian. *SOA with REST*. Pearson Education, Inc., 2013.

[2] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., 1st edition, 2015.

[3] Richard Rodger. *The Tao of Microservices*. Manning Publications Co., 2018.

[4] Eberhard Wolff. *Das Microservices-Praxisbuch*. dpunkt.verlag GmbH, 1st edition, 2018.

Referenced Web Resources

[5] A Guide to NumPy/SciPy Documentation. https://numpydoc.readthedocs.io/en/latest/format.html (accessed May 08, 2018).

[6] Free Documentation Licence (GNU FDL). http://www.gnu.org/licenses/fdl.txt (accessed March 01, 2018).

[7] Geomonitoring solutions by GEOINFO Vermessungen. https://www.geoinfo.ch/loesungen/detail/rc/Solution/sa/show/s/geomonitoring.html (accessed February 28, 2018).

[8] Leica deformation monitoring brochure. https://leica-geosystems.com/-/media/files/leicageosystems/products/brochures/leica_deformation_monitoring_bro.ashx?la=en (accessed February 28, 2018).

[9] Modular Applications with Blueprints. http://flask.pocoo.org/docs/0.12/blueprints/ (accessed March 07, 2018).

[10] OpenAPI Specification. https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.1.md (accessed March 01, 2018).

[11] Chris Richardson. Pattern: Microservice Architecture. http://microservices.io/patterns/microservices.html (accessed March 06, 2018).

Index