Hochschule für Technik Stuttgart

University of Applied Sciences

Master of Science Programme
Photogrammetry and Geoinformatics
Master Thesis
Winter Term 2017/2018

Visualization and Analysis of E-bike Usage in 3D City Model by Integration of Heterogeneous Sensor Data

by

Thunyathep Santhanavanich

Supervisors:

Prof. Dr.-Ing. Volker Coors	(Hochschule für Technik Stuttgart)
Dr. Wolfgang Käfer	(Daimler TSS GmbH)

Visualization and Analysis of E-bike Usage in 3D City Model by Integration of Heterogeneous Sensor Data

by

Thunyathep Santhanavanich

A dissertation presented in partial fulfillment of the requirements for the degree of Master of Science in the Department of Geomatics, Computer Science and Mathematics, Stuttgart University of Applied Sciences

Declaration

The following Master thesis was prepared in my own words without any additional help. All used sources of literature are listed at the end of the thesis.

I hereby grant to Stuttgart University of Applied Sciences permission to reproduce and to distribute publicly paper and electronic copies of this document in whole and in part.

Stuttgart, 28.02.2018

Thunyathep Santhanavanich

Approved by:

Prof. Dr.-Ing. Volker Coors

Acknowledgement

First of all, I would like to express my deep sense of thanks and gratitude to my supervisor Prof. Volker Coors, University of Applied Sciences Stuttgart for his support, enthusiasm, and suggestions. His guidance into the world of 3D geospatial visualization and IoT technologies have been extremely valuable for me during this work.

Additionally, I am deeply grateful to Dr. Wolfgang Käfer, Daimler TSS for giving me a chance to do this research in the i_city project in cooperation with Daimler TSS company. In addition, I express my warmest gratitude to all colleagues and staffs in the i_city project: Arno Hieber, Lara Kohn, Jan Eric Silberer, Alexander Ott, Habiburrahman Dastageeri, Martin Storz and Preston Rodrigues for their support and assistance.

Finally, I dedicate this thesis to my parents, Thanin Santhanavanich and Chortip Santhanavanich, for their genuinely support, love and encouragement.

Master Course Photogrammetry and Geoinformatics

Visualization and Analysis of E-bike Usage in 3D City Model by Integration of Heterogeneous Sensor Data

Abstract

According to the advances in information and communications technology, nowadays, the use of Internet of Things (IoT) has become a normal part of daily life as it allows interconnections among a wide variety of devices and sensors such as smartphones, smartwatches or any smart wearable devices, automobiles, or any object with a built-in sensor. However, these devices and sensors are developed by numerous different manufacturers which leads to the heterogeneity in the data management system. Consequently, a standard sensor protocol to sustain the interoperability among the various sensor systems is needed. In this research, the main contribution is building a sensor data management system for monitoring E-bike usages in the HFT i_city E-bike sharing project in the downtown area of Stuttgart city. The heterogeneous sensors in the project are including the SMART E-bike, the smartwatch Garmin Fenix X5, and the temperature data from OpenWeatherMap at the time this research is being conducted while the project is still open for more sensor devices and data sources. As for a scheme to tackle this challenge, three candidates of sensor protocols including 1) OGC SensorThings API, 2) OGC Sensor Observation Service, and 3) Advanced Sensor Data Delivery Service are implemented. Next, the performances by each sensor protocol are evaluated then the best-qualified protocol for the project is selected. Last, all sensor data stored remotely by the selected protocol is used to develop a 3D web-based application for a visualization and analysis of the E-bike usage in 3D city model.

Keywords: Smart Cities, 3D City Model, Wireless Sensor Network, Internet of Things, Heterogeneous Sensor, OGC Sensor Observation Services, OGC SensorThings API, ASDDS, 3D Web-based Application, Cesium, CityGML, CZML, glTF, Cesium 3D Tile, FME, 3DCityDB

Table of Contents

Table of Figures

Table of Tables

Abbreviations

E-bike	Electronic Bike
i_city	intelligent city
API	Application Programming Interface
IoT	Internet of Things
GML	Geography Markup Language
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
TCU	Telematic & Connectivity Unit
3DCityDB	The 3D City Database
OGC	The Open Geospatial Consortium
SWE	Sensor Web Enablement
SOS	Sensor Observation Service
ASDDS	Advanced Sensor Data Delivery Service
glTF	GL Transmission Format
HFT	Hochschule für Technik Stuttgart
CZML	Cesium Language
UML	Unified Modeling Language
XML	eXtensible Markup Language
REST	REpresentational State Transfer
SOAP	Simple Object Access Protocol
MQTT	Message Queuing Telemetry Transport

1. Introduction

The HFT Stuttgart joint research project entitled "i_city" (intelligent city) is working on solutions for central societal challenges of sustainable city development. The project is contained of six subprojects in different fields of action including urban planning, architecture, IT, energy, mobility, and finance with the main objective to develop the methods, services, and products for highly efficient energy, building, and mobility systems in urban areas. The 3D GIS tools, real-time sensor data, and Information and Communication Technologies (ICT) networking are employed for building an efficient plan, operating on sustainable energy and mobility system while allowing citizens to participate and influence the urban transition. [1, 2]

This master thesis research is a part of the "E-bike sharing project" which is one of the i_city subprojects in a field of mobility in cooperation with the Daimler TSS company. The mission of this subproject is to combine the E-bikes to the bike sharing system starting in the city of Stuttgart. [3] What makes E-bikes outstanding from any other normal bicycles for the Stuttgart's citizen is that they provide the electric motor support which helps them to easily commute in the mountainous terrain. Consequently, it could be a climate-friendly alternative choice for them to commute in the city area. Moreover, the E-bikes are equipped with the sensors that give useful information such as location, battery level, pedal force, motor support level, etc. [4] Accordingly, many researchers and developers can utilize this data in many different ways.

Since the IoT technology has been developing and improving for decades, its advancement allows these sensors to be sensed and controlled through the remote network and, eventually, leads to the development of a "Smart Cities" concept. [5] This concept is to utilize not only a single data source but many sensor data sources in order to give the informative knowledge to the citizens and solve the urban development challenges by using the right hardware, software, and technology platforms. [6] To apply this concept to the "E-bike sharing project", the capability to integrate other dynamic sensors such as wearable sensor devices, smartphones, etc., becomes necessary and allows researchers and developers to access various types of data. In 2017, a bachelor thesis on the emotion analysis during the E-bike usage period was conducted. In the thesis, the E-bike sensors and smart wearable devices were used, but there was no sensor network system to integrate the data from many different sensor devices together. Thus, it was very complicated and time-consuming to aggregate and utilize all the data. [7] In fact, different types of sensors provide different data formats, APIs, and data model structures. This controversy leads to the interoperability problem. [8] In order to solve this issue, the proper ways of communicating sensor locations, sensor and data parameters, and sensor instruction sets need to be addressed. [9, 10] According to OGC, managing the heterogeneous sensor data with an appropriate

sensor framework provides following benefits including (1) protects earlier investments, (2) prevents lock-in to specific products and approaches, and (3) allows for future expansion. [11]

In this research, it aims to study and compare the existing sensor frameworks or standards on their capabilities to integrate the heterogeneous sensor systems together and provide the data in an efficient way. The challenge of this work is to maintain the interoperability from the different development layers, for example, the communication protocols and data models. Also, to solve the issues of the irregular time-series data streams from numerous IoT devices, the sensor network must provide the capability to aggregate or interpolate the data over time. For the data collection, the E-bike usage data from twelve volunteers in the area of the Stuttgart city [7] are used. Three sets of sensor data used in the study are including the following data: 1) E-bike sensors, 2) the Garmin Smart Watch, and 3) the open-source temperature data in the city of Stuttgart from "OpenWeatherData". [12] Then, to verify of the sensor network interoperability on both server and client side, the data from the studied sensor network are used to implement the 3D web-based application for the visualization and analysis of the E-bike usage in the 3D city model.

2. State of the Art

2.1 Previous Works in Sensor Network Protocol

As the sensor technology, computer technology, and network technology are advancing together, the standards to link many diversity of technologies in an efficient way are needed. To address this, the Open Geospatial Consortium (OGC) has provided the Sensor Web Enablement (SWE) standards. With this standard, it allows users or developers to integrate many heterogeneous sensor systems with the accessible and manageable interface through the web. Many studies in topics of IoT networks use this standard as their sensor framework to manage diverse sensor data sources. The applications using the OGC SWE standards are, for example, traffic monitoring, airborne imaging, health monitor, air pollution monitoring, webcams etc. (Figure 1) [11]

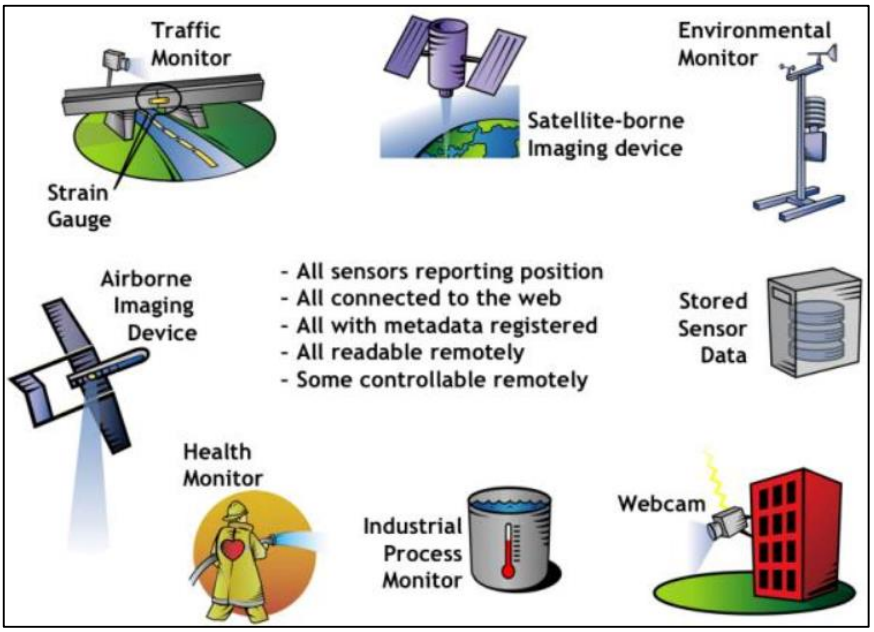


Figure 1: Example application of the OGC SWE standards, from [11]

In 2010, the OGC Sensor Observation Service (SOS), one of the OGC SWE standards, is used by the researcher for the US NASA ESTO/AIST Sensor Web program in a study in Geo-processing workflow-driven wildfire hot pixel detection under sensor web environment. The transaction Web Coverage Service was used between the OGC SOS and the Web Processing Service to feed the live EO-1 Hyperion data into fire hot pixel detection WPS. This study shows a good example of the implementation of the OGC SOS to satisfy the interoperability in the study. [13]

In 2016, researchers used OGC SOS to integrate the temporal air quality data and visualize with a 3D city model which using the open source implementation of SOS from 52°North. [14, 15] In

2017, this implementation version of 52°North SOS is successfully used again in many studies, for example, to develop a land surveying measurement model [16] , to manage the heterogeneous marine sensors [17] and to manage the large dataset of water quality monitoring system by the United States Environmental Protection Agency. [18] Additionally, some organizations have used this sensor standard to measure the real-world data, for example, PEGELONLINE uses various services to publish raw daily values of different water parameters of internal and coastal levels of the federal waterways up to a maximum of 30 days retroactively. [19]

According to the previous work related using the OGC SOS, many researches and studies have successfully used this standard to provide the interoperability among the heterogeneous sensor system. However, in some research, it is stated that using this standard still have a difficulty in handling the sensor data from SOS which provided in a standard-based SOAP style and XML-based encoding [20] Until 2016, the OGC SensorThings API had been approved as a new standard protocol to solve the same issue of heterogeneous sensor systems based on the OGC Sensor Web Enablement (SWE) standards and the ISO/OGC Observation and Measurement data model [5, 21] Many researchers and organizations have implemented this standard as the sensor network protocol. In 2017, a research in the United Kingdom [22] has studied about sustainable interoperability and data integration for healthcare system using this OGC SensorThings API framework with the case study of an IoT-based system for emergency medical services [23]. Also, the OGC SensorThings API is implemented as a sensor network system in a project "Smart emission" which is a citizen sensor network using low-cost sensors that enables users to get the knowledge about the environmental quality, for example, air quality, noise load, vibrations, light intensities and heat stress. [24] Another project from HFT Stuttgart [25], OGC SensorThings API standard is successfully used to store, manage and retrieve data from many sensors in the different rooms measuring the temperature of the rooms over a period of time.

In 2015, SensorThings API is investigated and compared with other three standard protocols including OGC PUCK over Bluetooth, TinySOS, and SOS over CoAP. As a result of this research, it is found out that TinySOS and OGC PUCK have the advantage in the ROM memory usage while the SOS over CoAP and the SensorThings API show the advantage in the minimum request and response size. [8]. In 2017, SensorThings API is used to develop the application dashboard to monitoring the water quality including temperature, turbidity, free chlorine, pH, and E. coli in the city of Vancouver. [26]

Additionally, the example use case of the SensorThings API from the SensorUp company [27] illustrates the usage of the SensorThings API with LeftletJS [28] showing the visualization of the real-time availability of the bike station in bike sharing program in the city of Toronto which is a good example for the IoT application in a "bike-sharing" concept. The following figure shows the bike station in the city of Toronto on the left and the 2D map application showing the bike station position with the

status of available bikes. (Figure 2) [29] It shows the capability of the SensorThings API that it can query the observation data from the several sensors at a city level in a specific time and location.

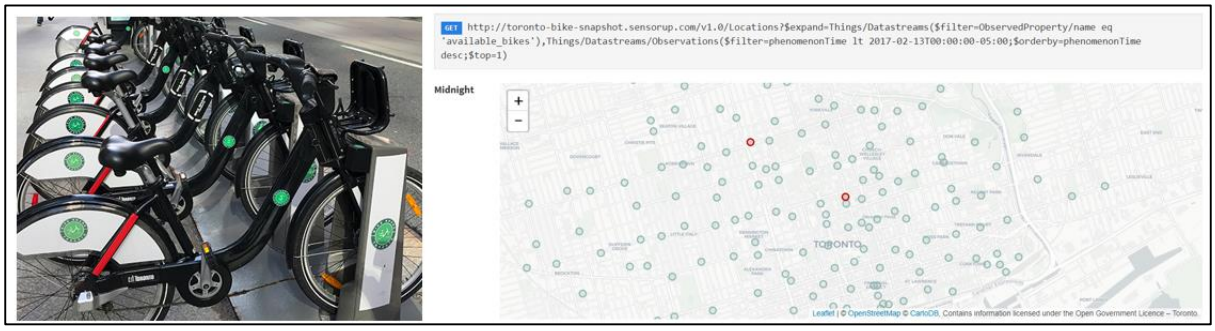


Figure 2: (Left) The bike station in the city of Toronto, (Right) The 2D map application showing the bike station position with the status of available bikes.

In 2016, a project from HFT Stuttgart in the topic of "Visualisierung von Sensordaten" aims to obtain and visually represent the sensors data through a web service with an option for users to request for an aggregated result of the sensor data, for example, to query for a mean or average value of the sensor results over a specific period of time. However, there is currently no open sensor standard that provides such a capability. For that reason, the sensor protocol "Advanced Sensor Data Delivery Service - ASDDS" is created as a new protocol to provide an aggregation function over the web service. [30]

2.2 Feature Comparison of the Sensor Protocols

To conclude the findings from previous works in several sensor protocol standards, the technical feature comparison of these three protocols are shown up in (Table 1). This table is the extended table based on the comparison chart between SensorThings API and SOS in a previous work [31]. Overall, a large number of researchers successfully used the OGC SOS as their sensor management system as an interface for accessing sensor data and metadata since the first version of SOS has been approved in 2007. Still, some research finds this sensor framework is difficult in application development stage. [20] In contrast to the OGC SensorThings API, it is the newest OGC standard for managing the sensor data and still has a low number of studies based on this standard [32], and this standard is developed based on the same OGC SWE with the improvement concept that overcomes the OGC SOS in many ways [31], for example, it supports the pagination and MQTT, a powerful machine-to-machine (M2M) connectivity protocol. Also, the main encoding of OGC SensorThings API in JSON with REST bindings which are more friendly and simpler for developers to understand, develop and expand the IoT application and keep its extensibility and maintainability. [33] However, both standards still have some gap in the aggregation function support which is the main reason that the sensor protocol

"Advanced Sensor Data Delivery Service" is developed in a student project in HFT Stuttgart in 2016. [34]

Table 1: Features comparison of three different sensor protocols

Protocols: / Features:	OGC SensorThings API	OGC SOS	ASDDS
Encoding	JSON	XML + (JSON in 52°North SOS 4.4)	JSON
Architectural Style	Resource Oriented Architecture	Service Oriented Architecture	Resource Oriented Architecture
Binding	REST	SOAP	REST
Insert new Sensors and Observation results	HTTP POST	SOS specific interface: InsertSensor() / InsertResultTemplate() / InsertResult() /	HTTP POST
3D Location of a Moving Sensor	Supported directly through [Location and Historical Location] Entities	Supported by SOS specific interface: Spatial observation()	Not supported directly [Input as 3 Observation Values : Lat/Long/Height]
Deleting existing sensors	HTTP DELETE	OGC SOS Specific interface: DeleterSensor()	Not supported
Deleting existing Observation	HTTP DELETE	Not supported	Not supported
Pagination	$top, $skip, $nextLink	Not Supported	Not Supported
Pub/Sub Support	MQTT	Not Supported	Not Supported
Sensor Metadata	Supported (OGC O&M specification)	Supported (OGC O&M specification)	Supported (Name and Unit)
Updating properties of existing sensors or observations	HTTP PATCH and JSON PATCH	Supported	Not Supported
Linked data support	JSON-LD	Not Supported	Not Supported
Request multiple O&M Entities (e.g., FeatureOfInterest and Observation) in one request/response	Using $expand	Not Supported	Not Supported
Aggregation Function over the request	Not Supported	Not Supported	Supported (Only mean value possible in this version)

Accordingly, it can be stated that each protocol has its own advantages and disadvantages and all of them have been proved in the realistic projects on their interoperability and capability of heterogeneous sensors management. This thesis research will study and compare all three mentioned protocols including 1) OGC SensorThings API, 2) OGC Sensor Observation Service (SOS), and 3) Advanced Sensor Data Delivery Service (ASDDS) to find the best-qualified standard for managing dynamic E-bike usage data.

3. Background

3.1 Sensor Protocols

To find the most qualified sensor data management framework for integrating the heterogeneous sensor systems in i_city E-bike sharing project, based on the finding of the previous work in (Chapter 2) the following candidate protocols are studied and implemented: 1) OGC SensorThings API, 2) OGC Sensor Observation Services, and 3) Advanced Sensor Data Delivery Service (ASDDS).

3.1.1 OGC SensorThings API

The OGC SensorThings API is an OGC community standard providing an open and unified framework to interconnect IoT devices, data, and applications. It allows the IoT devices and applications to CREATE, READ, UPDATE, and DELETE IoT data and metadata through a HTTP request. From February 2016, the SensorThings API standard is approved to be a part of an open international standards OGC Sensor Web Enablement (SWE) suite. The OGC SWE is used in many international organizations such as NASA, NOAA, USGS, Natural Resources Canada, and many others, including many private sector companies. [11] Accord to Steve Liang [35], one of the authors of SensorThings API standards, he stated that using this standard as the data model and the interface for access the sensor data provides the following benefits: 1) it permits the proliferation of new high-value services with lower overhead of development and wider reach, 2) it lowers the risks, time and cost across a full IoT product cycle, and 3) it simplifies the connections between devices-to-devices and devices-to-applications. For the relation between the data model of this standard, the UML diagram is shown in (Figure 3).

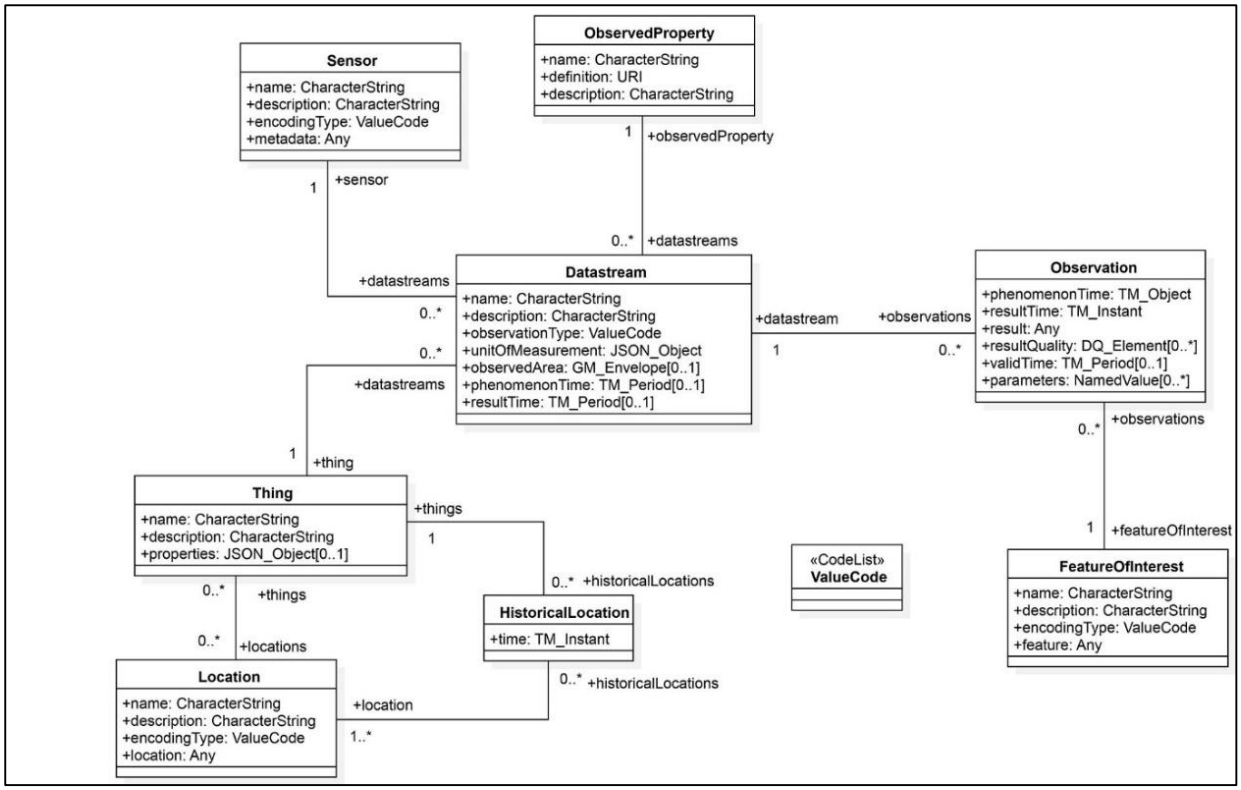


Figure 3: UML Data Model of SensorThings API, figure from [5, 35]

To describe the data model of this standard, the physical object is modeled as a "Thing" which can be referred to each E-bike in the project. And each "Thing" is related to the "Location" entity which collecting the latest location. When the "Location" of the "Thing" updates, the old "Location" is collected in the "HistoricalLocation" entity instead. While the "Sensor" entity is referred to each sensor equipped with the "Thing" which means each "Thing" can have one or more "Sensor" entity (or entities). Each "Datastream" entity observes the unique "ObservedProperty" entity, "Sensor" entity, and "Thing" entity. Finally, the result sent from the sensor is described as the "Observation" entity which each "Observation" can be matched to the prepared unique "Datastream" entity and observes the unique "FeatureOfInterest" entity. With all these relationships, they provide the flexibility to add more sensors, physical objects, and features of interest into the sensor network without any conflict. [5, 36]

All the communications with the data service followed the REST-like architecture. The following figure shows the example of the HTTP GET request and response to the OGC SensorThings through the Internet. (Figure 4) [8, 37]

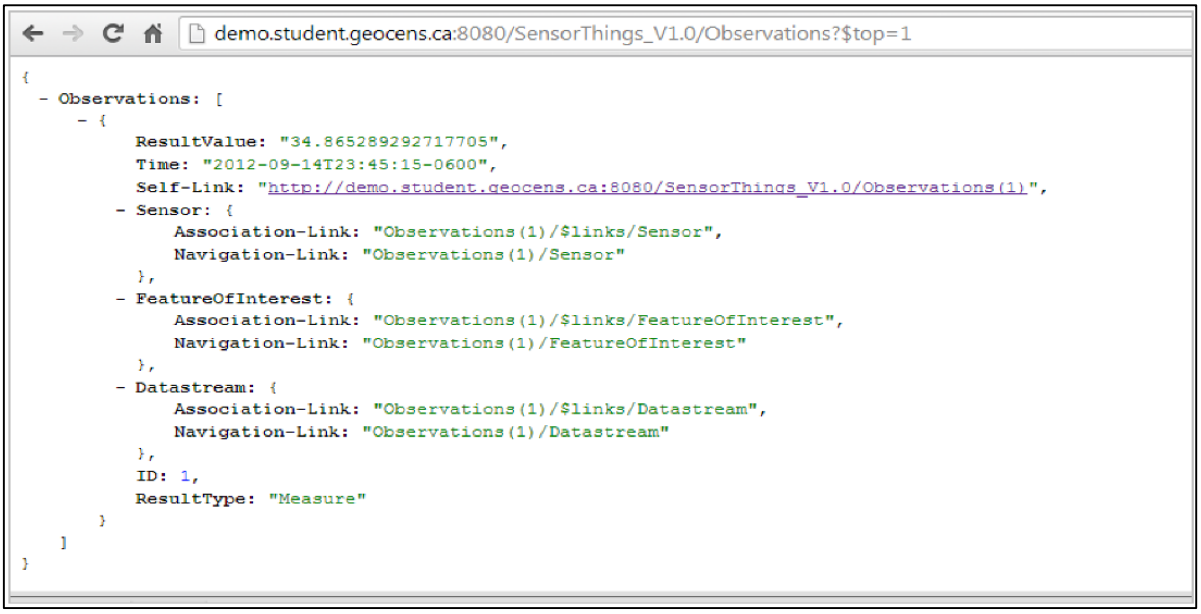


Figure 4: the example of the HTTP GET request and response to the OGC SensorThings through the Internet [8]

To compare with the OGC Sensor Observation Services (SOS) sensor network standard, the SensorThings API's data model is also based on OGC/ISO Observations and Measurements [38] and both are parts of the OGC SWE suite so that they can easily interoperate with each other. The main difference is that SensorThings is RESTful, uses the efficient JSON encoding, adopts the OASIS OData URL pattern and query options, and supports the ISO MQTT messaging protocol. [32] The main advantage of the SensorThings API is that it is an extremely lightweight IoT Data Interoperability standard that uses both MQTT and HTTP REST.

According to the seminar about the SensorThings API [39], the steps for data analytics or data mining are 1) Data Cleaning, 2) Data Integration, 3) Data Selection, 4) Data Transformation, 5) Data Mining, 6) Pattern Evaluation, and 7) Knowledge Presentation. It is stated that the first four steps mentioned can be done simply with SensorThings API connecting with many different systems which are normally very time to consume to be implemented.

3.1.2 OGC Sensor Observation Services (SOS)

The Sensor Observation Service (SOS) is a web service to query real-time sensor data and sensor data time series and is part of the OGC Sensor Web Enablement (SWE) standards. [40] It enables developers to make all types of sensors, transducers and sensor data repositories discoverable, accessible and useable via the Web. [11] According to the OGC website [41], the Sensor Observation Services (SOS) standard is applicable to use cases in which sensor data needs to be managed in an interoperable way. This standard defines a Web service interface which allows querying observations, sensor metadata, as well as representations of observed features. Further, this standard defines means to register new sensors and to remove existing ones. Also, it defines operations to insert new sensor observations. This standard defines this functionality in a binding independent way. According to the standard document [42] two bindings are specified: a KVP binding and a SOAP binding. The SOS is designed to provide access to observations. For the SOS 2.0, these observations are by default encoded conformant to the Observations and Measurements 2.0 (O&M 2.0) standard. [43] An Observation provides a result whose value is an estimate of a property of the observation target, the feature of interest; i.e. an observation is a property-value-provider for the feature of interest. An instance of an Observation is classified by its phenomenon Time, FeatureOfInterest, ObservedProperty, and the procedure used. The procedure is usually a sensor but can also be for example a computation or post-processing step. [42] The basic relation between the observation and other entities is shown in the model depicted in (Figure 5).

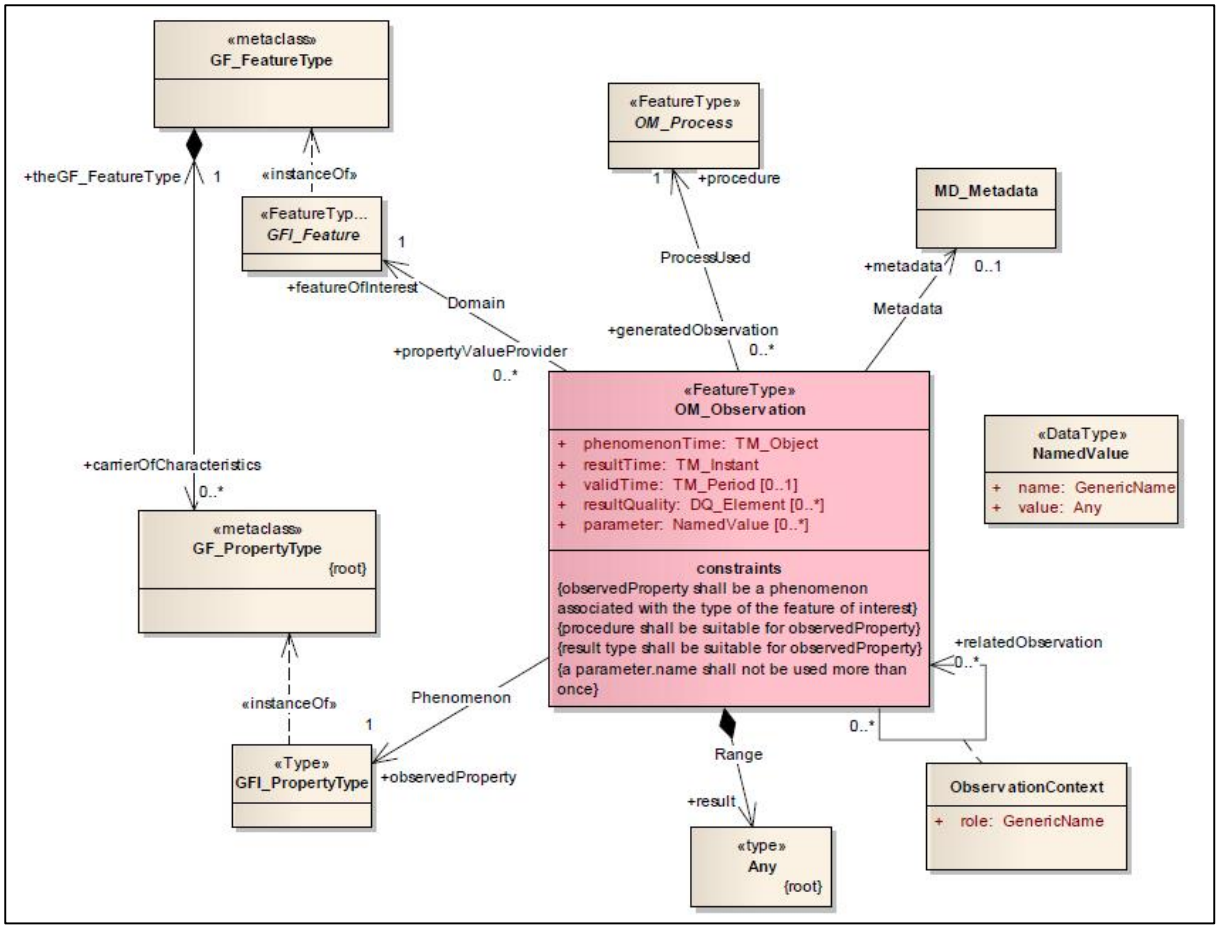


Figure 5: OGC Observations and Measurements Basic Observation Model

For example, the figure below (Figure 6) [43] shows the example of the sensor measuring the wind speed in the area of Mississippi. The "Feature of interest" is a representation of a real-world object which in this case is an area of "Mississippi". The "Observed property" is the wind speed. The "Phenomenon time" is the time when the wind speed result applies while the "Result time" is the time when the wind speed is observed. The "Procedure" in this case is an instance of a process which is thermometer "t_234". And the "Result" is the result value of the wind speed. For the usage of this standard, there is an example SOS service available online called "PEGELONLINE Sensor Observation Service" [44] which publishes raw daily values of different water parameters of internal and coastal levels of the federal waterways up to a maximum of 30 days retroactively and it is still running up-to-date and open by everyone to test the service by using the PegelOnline SOS TestClient [45]. The following figure (Figure 7) shows the example of the "GetObservation" operation to get the latest observation value of the measured water temperature result at the sensor station "Wassertempeatur-Nalje_Siel_126001".

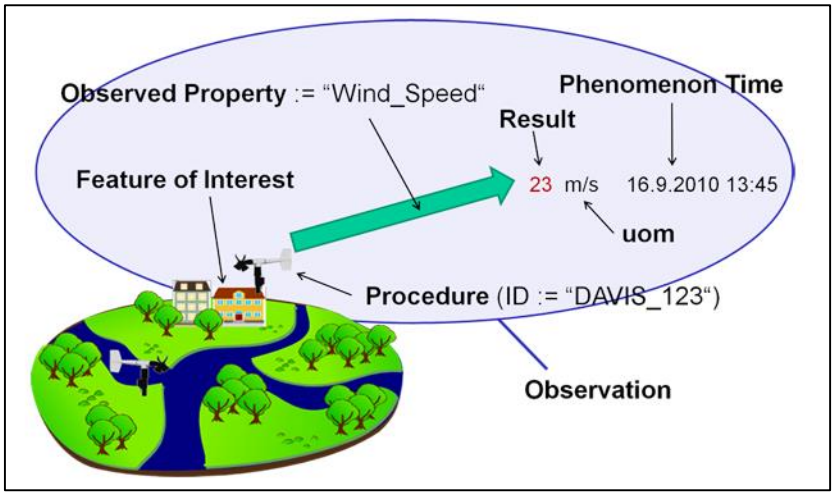


Figure 6: Example of the central terms of the OM 2.0 model [43]

```
<om:member>
    <om:Observation>
        <om:samplingTime>
            <gml:TimePeriod xsi:type="gml:TimePeriodType">
                <gml:beginPosition>2018-01-25T13:44:00.000Z</gml:beginPosition>
                <gml:endPosition>2018-01-25T13:44:00.000Z</gml:endPosition>
            </gml:TimePeriod>
        </om:samplingTime>
        <om:procedure xlink:href="Wassertemperatur-Nalje_Siel_126001"/>
        <om:observedProperty>
            <swe:CompositePhenomenon gml:id="cpid0" dimension="2">
                <gml:name>Result Components</gml:name>
                <swe:component xlink:href="http://www.opengis.net/def/property/OGC/0/SamplingTime"/>
                <swe:component xlink:href="Wassertemperatur"/>
            </swe:CompositePhenomenon>
        </om:observedProperty>
        <om:featureOfInterest>
            <gml:FeatureCollection>
                <gml:featureMember>
                    <sa:SamplingPoint gml:id="Nalje_Siel_126001" xsi:schemaLocation="
http://www.opengis.net/sampling/1.0 http://schemas.opengis.net/sampling/1.0.0/sampling.xsd">
                        <gml:description>NOT_SET</gml:description>
                        <gml:name>Nalje Siel - Kilometer: 70</gml:name>
                        <sa:sampledFeature xlink:role="urn:x-ogc:def:property:river"
xlink:href="http://pegelonline.wsv.de/webservices/gis/gdi-
sos?REQUEST=getFeatureOfInterest&service=SOS&version=1.0.0&featureOfInterestID=OSTE"/>
                        <sa:position>
                            <gml:Point>
                                <gml:pos srsName="urn:ogc:def:crs:EPSG::4326">53.83357371129295
9.034589157098823</gml:pos>
                            </gml:Point>
                        </sa:position>
                    </sa:SamplingPoint>
                </gml:featureMember>
            </gml:FeatureCollection>
        </om:featureOfInterest>
        <om:result>
            <swe:DataArray>
                <swe:elementCount>
                    <swe:Count>
                        <swe:value>1</swe:value>
                    </swe:Count>
                </swe:elementCount>
                <swe:elementType name="Components">
                    <swe:DataRecord>
                        <swe:field name="SamplingTime">
                            <swe:Time definition="http://www.opengis.net/def/property/OGC/0/SamplingTime">
                                <swe:uom xlink:href="http://www.opengis.net/def/uom/ISO-8601/0/Gregorian"/>
                            </swe:Time>
                        </swe:field>
                        <swe:field name="FeatureOfInterest">
                            <swe:Text definition="http://www.opengis.net/def/property/OGC/0/FeatureOfInterest"/>
                        </swe:field>
                        <swe:field name="Wassertemperatur">
                            <swe:Quantity definition="Wassertemperatur">
                                <swe:uom code="°C"/>
                            </swe:Quantity>
                        </swe:field>
                    </swe:DataRecord>
                </swe:elementType>
                <swe:encoding>
                    <swe:TextBlock decimalSeparator="." tokenSeparator="," blockSeparator=";"/>
                </swe:encoding>
                <swe:values>2018-01-25T13:44:00.000Z,Nalje_Siel_126001,4.0;</swe:values>
            </swe:DataArray>
        </om:result>
    </om:Observation>
</om:member>
```

Figure 7: The example of the "GetObservation" operation to get the latest observation value of the measured water temperature result at the sensor station "Wassertempeatur-Nalje_Siel_126001".

3.1.3 Advanced Sensor Data Delivery Service (ASDDS)

The ASDDS sensor protocol is created in a project "Visualisierung von Sensordaten" as mentioned in (Chapter 2.1) to provide the sensors data through a web service together with the aggregate function. The case study in this project is to measure the temperature of a room in C° or the absorbed power of a heater in kW/h. In addition to the pure representation, it should also be possible to vary the temporal resolution of the graphs. Additionally, the setting of the time interval and the selection of a sensor should be possible interactively. [30]

The REST API and the web application for visualizing the data are implemented to match the API and the client optimally. To visualize similarities and relationships between the different sensors, a line diagram is used for a detailed view. The following figure shows the example of the ASDDS Client application with sensor data from HFT Stuttgart. (Figure 8) [30]

Figure 8: ASDDS Client application with sensor data from HFT Stuttgart. [30]

For the implementation of this work, on the client side, JavaScript is used with the Highcharts library [46] to visualize the sensor data. The client communicates via JSON with the REST interface implemented by us in the Java Framework Dropwizard [47], which provides the data in the appropriate aggregation. The database system used is a PostgreSQL database. [48] An additional script is implemented in Python that imports the existing data in table format to the database via the REST port with the specified observation parameters as in the list in (Table 2). [30]

Table 2: ASDDS Observation parameters

Parameter	Definition
Sensor Name	Name of the sensor as a string.
Unit	Unit of the sensor.
Address	Address of the sensor as a string.
Time Interval	The start and the end time in UNIX-Timestamp format
Aggregation	Aggregation of min, max or average value per day

3.2 E-bike Usage Field Data Collection

The E-bike usage data in this research are based on the volunteer E-bike usages from the other research in HFT i_city E-bike sharing project from the Business Psychology department, HFT Stuttgart. According to the research of Lara Kohn, in October 2017 [7], her work is to investigate relationships of emotional experience and various physiological parameters of E-bike riders and to identify psychophysiological markers. The study refers to a target group of twelve female students aged 20 to 25 years. All participants take the similar bike route around 10 kilometers long with the starting point from HFT Stuttgart to the Killesberg direction as shown in (Figure 9).

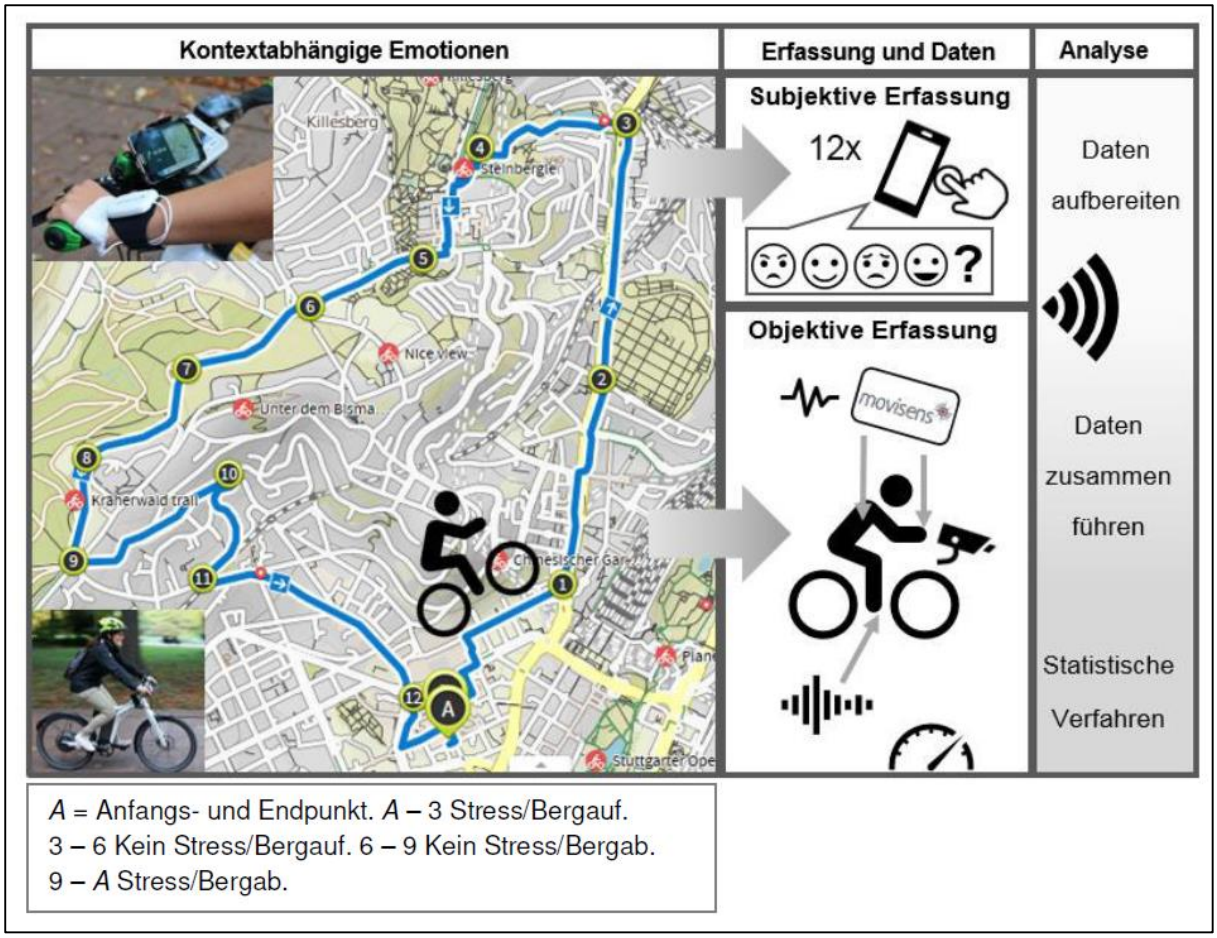


Figure 9: E-bike route in the HFT i_city research, figure from research of Kohn, L. [7]

For her research, the sensor devices from Movisens are used to measure the health data of users. They can capture high-resolution electrocardiography (ECG), electrodermal activity (EDA), and physical activity data to record and analyze psycho-physiological parameters. [14] The Movisens products in this research are "EdaMove 3" and "EcgMove 3". The EdaMove 3 (Figure 10) provides the comprehensive tool for recording and analyzing Electrodermal (Galvanic Skin Response) and physical activity. [49] And the EcgMove 3 (Figure 11) is a psycho physiologic ambulatory measurement system for the assessment of ECG and physical activity. [50]

Figure 10: Movisens EdaMove 3 [49]

Figure 11: Movisens EcgMove 3 sensor [50]

After Lara Kohn's research ended, Jan Eric Silberer took part in continuing her research work by increasing the number of participants and adding smartwatch as a sensor device starting in November 2017. The smartwatch model using in this research is Garmin fēnix® 5X. (Figure 12) It is a multisport GPS watch with full-color mapping, routable cycling maps, and other outdoor navigation features. It contains built-in navigation sensors include GPS and GLONASS capability to track in more challenging environments than GPS alone as well as 3-axis compass, gyroscope and barometric altimeter. With the Wrist-based Heart Rate, it lets users monitor their heart rate without wearing a chest strap. In addition to counting steps and monitoring sleep, the watch uses heart rate to provide calories burned information and quantify the intensity of their fitness activities. [51]

Figure 12: Garmin Smart Watch fēnix® 5X

3.3 Current Sensor Network Architecture

The SMART E-bikes (Figure 13 - left) are used in the i_city project. They are equipped with the BioniX sensors which provide the E-bike user data and then those data are sent to the data server through the Telematic & Connectivity Unit (TCU) (Figure 13 - right). The TCU plays a central role in this work to communicate and transfer the data between the vehicle and the backend system. It is originally used by "car2go", the carsharing service of Daimler AG to ensures that customers can start and end the rental of a vehicle. [52] The TCU is installed with a mobile modem to connect to the Internet for making an interaction with the server network. [4, 53]

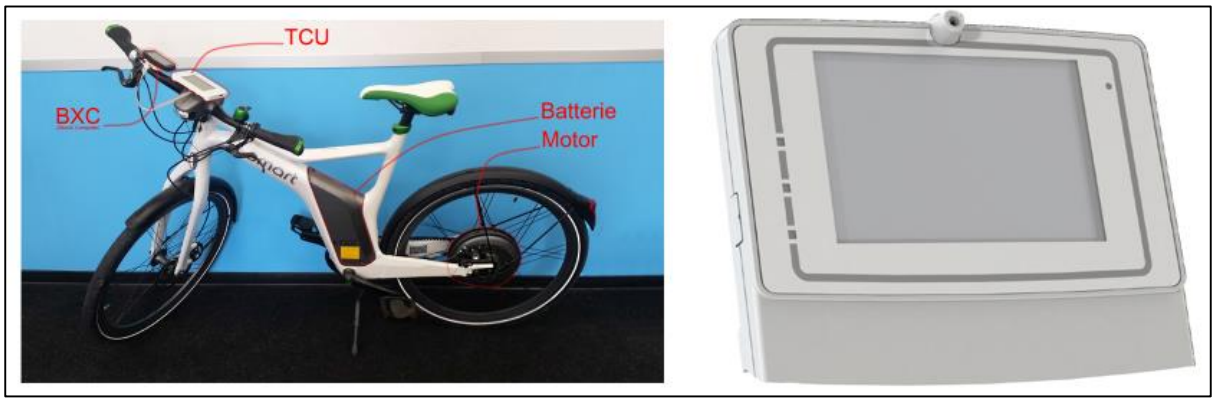


Figure 13: (left) Showing the Smart E-bike with the important components, (right) Telematic & Connectivity Unit[4]

Related to the previous bachelor research in i_city E-bike sharing project by Alexander Ott [4], the sensor platforms have been built to manage the sensor data from E-bike dynamically. The overview of the architecture structure of this network is shown in (Figure 14). The sensor data sent from the TCU is firstly received by a "Bartender" via MQTT or GSM. The sensor platform called "VSS" or "Vehicle Status Service" is built to receive the latest updated E-bike data from Bartender and create a service of the latest status of each vehicle through a REST API in the form of a JSON object string. However, the "VSS" service itself does not save the historic vehicle status. But, it allows users to deposit an Internet Protocol (IP) address or a Uniform Resource Locator (URL) to subscribe the E-bike status data. To compensate the issue that VSS needs for a static IP address, another sensor platform called "STASH" is built up to collect the historical E-bike data and make them available for users to request this data. However, the "STASH" is designed in contrast to the bartender which is expected to work reliably at a small load of the vehicles in the system. [4]

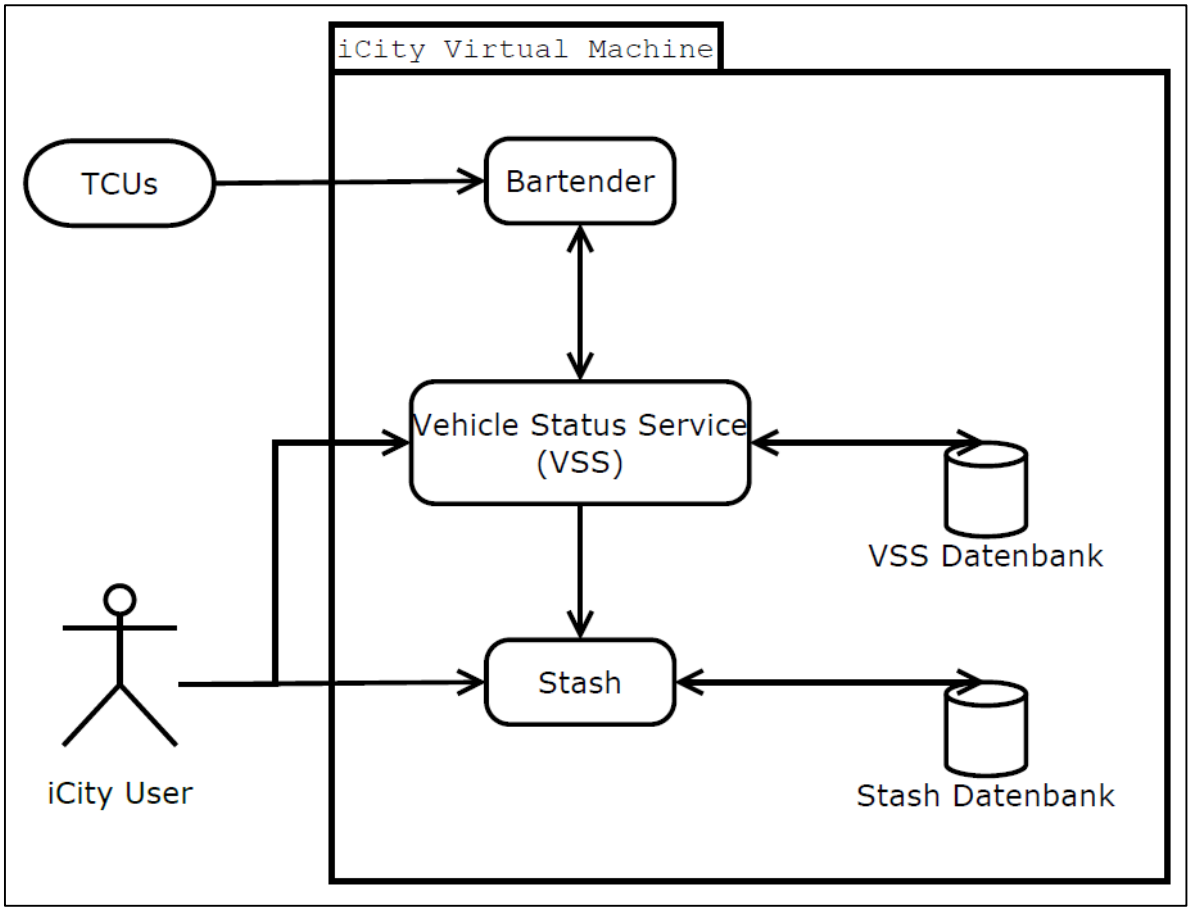


Figure 14: The current Sensor Network Architecture for Smart E-bike

In order to securely get the E-bike sensor data for i_city research, the "STASH" system is planned to be installed internally in HFT Stuttgart. While the "STASH" is still in the development process, users in HFT Stuttgart can access the data through the temporary solution by the FTP connection to get the daily update of the E-bike usage dataset as a log file. The more details on E-bike data connection will be described in (Chapter 4.3.1: Network Architecture)

3.4 3D Web-based Application

After the sensor protocol is implemented, the 3D web-based application is built up to verify the operability of the data from numerous sensors to the client. According to the research from Fraunhofer Institute for Computer Graphics Research in 2015, [54] they had implemented and compared three Open-Source frameworks for 3D Web-based visualization: X3DOM, three.js, and Cesium. The result from this study shows that the Cesium is the only one Open-source framework that fully supports geospatial applications with the built-in geospatial coordinates system. While the Three.js has no geospatial support and need an extension to enable the development of the application with geospatial data.

Moreover, according to one of the Cesium application showcase, "Cycling the Alps" [55], it is able to show the routes of the historical cycling climbs in the Alps mountain with the features to playback the bike usage on the 3D terrain as a time-series data linking with the clock and time slider in the application. (Figure 15) The data of the bike routes are originally the KMZ format, then they are transformed into the CZML (Cesium Language) file format [56] which allow users to specify the time-tagged data which the client will interpolate over to compute the value of the property at any given time.

Figure 15: Cycling the Alps Application [57]

Accordingly, the open-source 3D globe application that is suitable for this research is Cesium library. [58] It is an open-source JavaScript library which provides many examples and tutorial for new users. Importantly, it is supported by the dynamic geospatial visualization. There are many advantages of using the Cesium library for this research to make a visualization of the E-bike path. Firstly, it is supported by the 3D terrain which Analytical Graphics, Inc. has provided free terrain called "STK World Terrain" for use in any application and for any purpose. [59] The example screenshot of the STK World Terrain showcase showing the area of the Mount Everest in Cesium application [60] is shown in (Figure 16 - left). This functionality is very helpful to simulate the E-bike usage on the realistic 3D environment. Secondly, it supports 3D path which is the polyline path of any object defined by the motion of an object over time. [61] For instance, the example of GPS flight data path in Cesium Sandcastle application [62] is shown in (Figure 16 - right). For this research, the functionality of showing object motion over time is significantly needed to creating a time-dynamic E-bike path simulation.

Figure 16: (left) screenshot of the Cesium Application showing 3D STK World Terrain, (right) screenshot of the Cesium Application showing CZML Path of GPS flight data

Moreover, the Path data in Cesium has several options to interpolate the position of the moving object over time including three possible algorithm of interpolation methods including "Linear approximation", "Lagrange Polynomial Approximation", and "Hermite Polynomial Approximation" which developer can also freely choose the degree of interpolation in case of “Lagrange” and “Hermite” polynomial approximation. [63] The example of three interpolation GPS flight paths in Cesium application is shown in (Figure 17). This functionality can be applied later to simulate the E-bike usage data. To limit the study in this research, there is no comparison among the different approaches and the “Lagrange Polynomial Approximation” is used in the final application.

Figure 17: Screenshot of the Cesium Application showing three different methods of Path interpolation (left: Linear Approximation, middle: Lagrange Polynomial Approximation, right: Hermite Polynomial Approximation)

3.5 3D City Model

The 3D city model used in this research is a sample data in LoD 2 from Landeshauptstadt Stuttgart in the downtown area of Stuttgart city [64]. This dataset is based on the CityGML format which is an open standardized data model and exchange format to store digital 3D models of cities and landscapes. It is implemented as an application schema for Geography Markup Language Version 3.1.1 (GML3), the extendible international standard for spatial data exchange issued by the Open Geospatial

Consortium (OGC) and the ISO TC211. [65] The FME Inspector 2017 [66] is used to make the sample visualization to investigate this dataset in both 2-dimension (Figure 18) and 3-dimension (Figure 19)

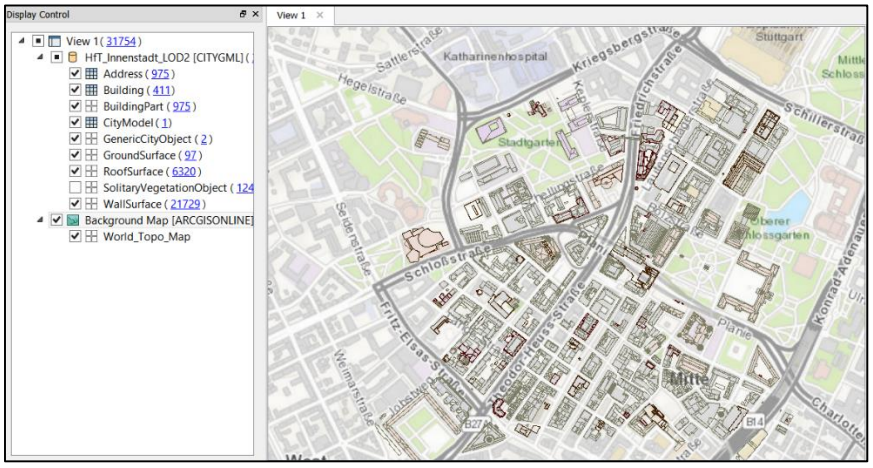


Figure 18: CityGML data in the downtown area of Stuttgart in 2-dimension with the ArcGIS Online topology basemap by FME Inspector 2017 Software [64, 66, 67]

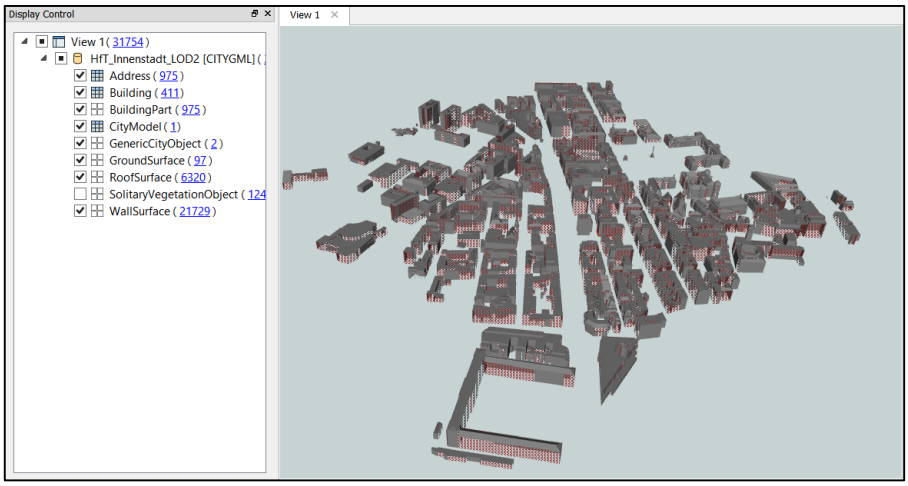


Figure 19: CityGML data in the downtown area of Stuttgart in 3-dimension by FME Inspector 2017 Software [64, 66]

As the 3D city model in CityGML format is not currently supported to visualize on the Earth's web-based application directly, the processes to transform this CityGML data into other support format are needed. For the Cesium application, the supported 3D city model formats are glTF (GL Transmission Format) and Cesium 3D Tile. The available tools that have the capability to do this transformation into those formats are including (1) 3D City Database, (2) FME Desktop 2017, and (3) Georocket GeoToolbox. [66, 68, 69] The usage of these three tools will be explained in (Chapter 4.4 : Preparing 3D City Model).

3.5.1 glTF

The glTF is file format for the 3D scenes and models using the JSON standard developed by the Khronos Group 3D Formats Working Group. [70, 71] It can provide the efficient transmission and loading of 3D scenes and models by applications with the minimization in both the size of 3D assets and the runtime processing. The example of visualization of the CityGML features in a glTF format with the 3DCityDB-Web-Map-Client is shown in (Figure 20). [72]

Figure 20: The example of visualization of the CityGML features in glTF format with the 3DCityDB-Web-Map-Client [72]

3.5.2 *Cesium 3D Tiles*

As the final application is built with Cesium library, the common 3D city model format that can be used to make a visualization on Cesium globe is "Cesium 3D Tiles". It is an open specification for streaming 3D geospatial datasets which are open, optimized for streaming and rendering, interactive, styleable, adaptable, flexible, heterogeneous and precise. [73] It can be used to stream 3D content, including buildings, trees, point clouds, and vector data. The following figure shows the example of the visualization of the 3D city model in the area of New York City. (Figure 21)

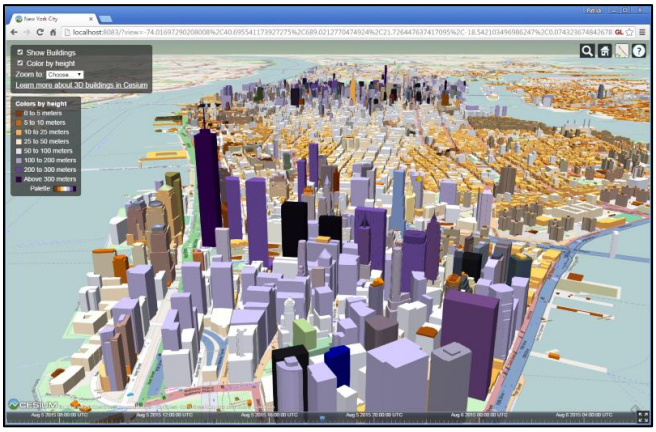


Figure 21: The Cesium application shows the OpenStreetMap buildings in New York City.

4. Methodology

4.1 Research Processes

In this research, overall processes can be summarized as in (Figure 22) which are divided into five main steps which are related to the final architecture of the project shown in (Figure 23). In the first step, "① Implementation of the Sensor Network" (Chapter 4.2), the three selected sensor protocols are studied and implemented in the local server which is including 1) OGC SensorThings API, 2) OGC SOS, and 3) Advanced Sensor Data Delivery Service (ASDDS). The open source implementation versions of the OGC standards SensorThings API and SOS are available on the online repository source like Github with the usage documentation. [14, 74] On the other hand, the implementation version of ASDDS and its guideline is provided from HFT Stuttgart by a colleague who used this protocol. [34] The main goal of this step is to study and compare on their usages, limitations, capabilities or features in all sensor protocols.

Then, the next step "② Sensor Data Management" is to manage the heterogeneous sensor data from E-bike TCU, Garmin Smart Watch, and other sensor data sources. In this step, the data management is starting with network architecture to link the data from different sources into the sensor network built form the first step ①. Then, the data cleaning is needed as each sensor data sources provide the redundant fields or entities in a different way. After that, the data modeling is done to match with the sensor protocol specification according to the first step ①. Finally, the prepared data will be imported into the sensor network. On the third step, "③ Preparing 3D City Model Data" is to prepare the 3D city model data in the downtown area of Stuttgart city which is provided in the CityGML format. (More details in Chapter 3.5: 3D City Model) As the CityGML is not able to be visualized on the Cesium application in the 3D environment directly. Thus, the conversions of the CityGML into Cesium 3D Tile or glTF are needed. The possible tools to do this conversion are including 1) FME 2017, 2) Georocket GeoToolbox 1.0.2, and 3) 3DCityDB. On the fourth step, "④ Developing 3D Application using Cesium", the main idea is to build an application by utilizing the sensor data from the prepared sensor network which is done on step ① and ② together with the prepared 3D city model data from the third step, ③. The application will be built in 3D with the JavaScript library Cesium. The more details on why Cesium is chosen to use in the application development are explained on (Chapter 3.4: 3D Web-based Application). Then, on the last step, "⑤ Evaluation" is to evaluate this research on the different implemented sensor protocols and the different methods to preparing the CityGML to visualize as a 3D city model on Cesium application.

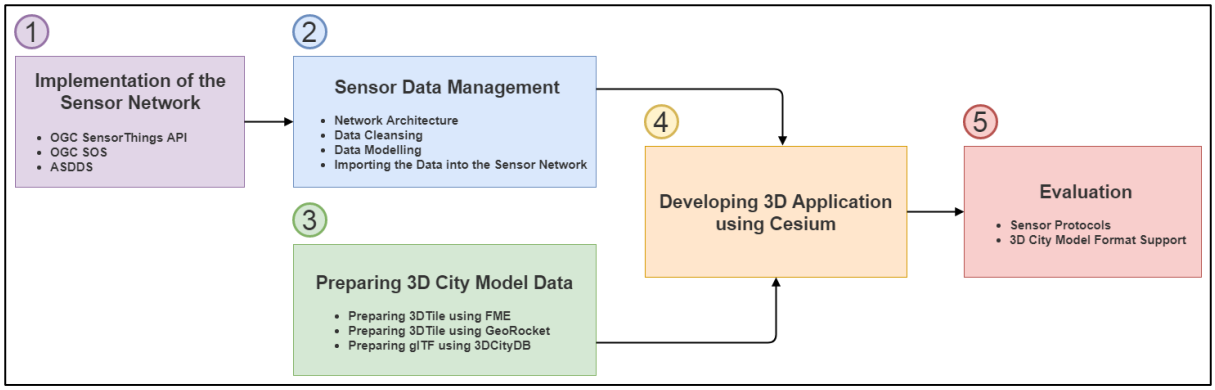


Figure 22: The overall research processes

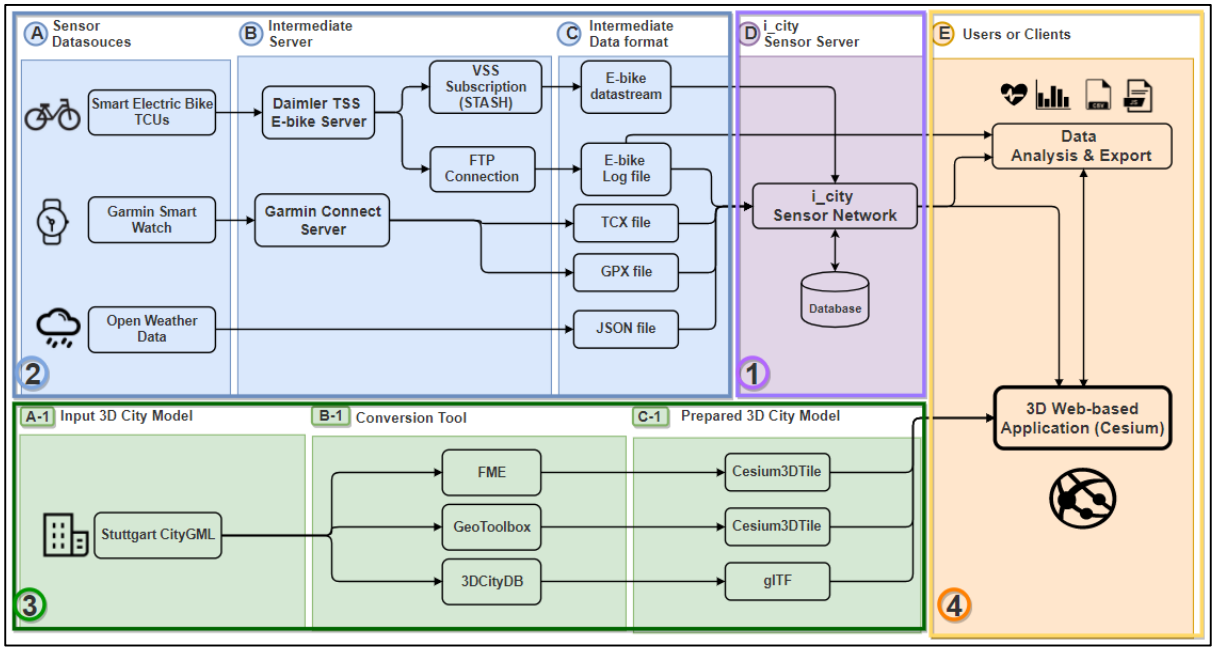


Figure 23: The overall architecture from the device layers to the client layers

4.2 Implementation of the Sensor Networks

4.2.1 OGC SensorThings API

To apply the OGC SensorThings API in this research, the server implementation of the OGC SensorThings API from Fraunhofer Institut IOSB is used. [74] The first step to use this server is to use the Apache Tomcat Maven 2.2 [75] to create a Web application Archive or WAR file from the root folder of this source code. Then, the Tomcat server must be installed on the machine that is used for receiving the sensor data. In this study, the default port 8080 of Tomcat server is used. Later on, the prepared SensorThings API WAR file is used to deploy on this server with the application name of "SensorThingsService". The simple structure of the SensorThings API server is shown in (Figure 24).

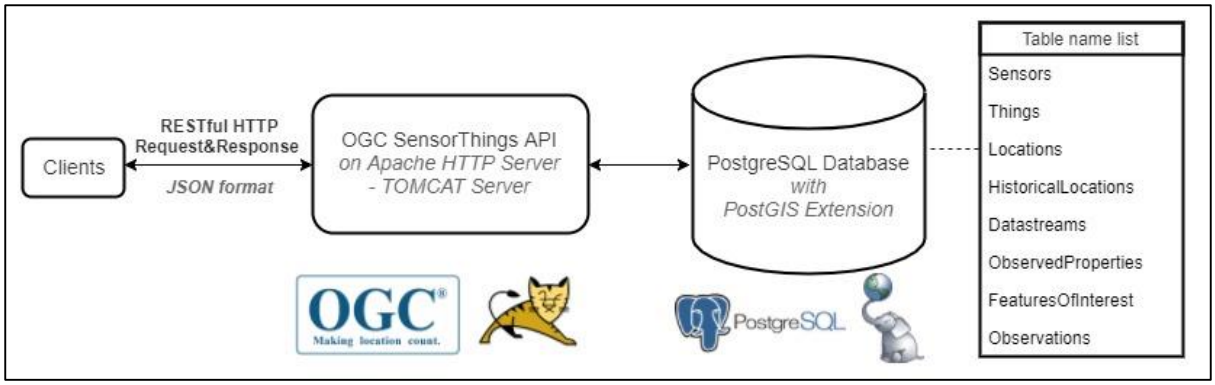


Figure 24: SensorThings API server structure

After the SensorThings API is successfully deployed on the machine, this application allows users to make a communication with RESTful HTTP method to make requests and get responses in JSON format via a SensorThings base resource path as the following URL pattern.

"http://[IP name/address]:[port]/[application name]/[version]/[entity name]"

Correspondingly, users can access to the SensorThings API server with a base resource path of "http://localhost:8080/SensorThingsService/1.0" on the web browser, the response to this "GET" request will be a JSON array showing the links to all the available accessible entity sets as in (Figure 25). To make a JSON array easy for understanding on the web browser, the browser Chrome with JSON Viewer extension is used. [76]

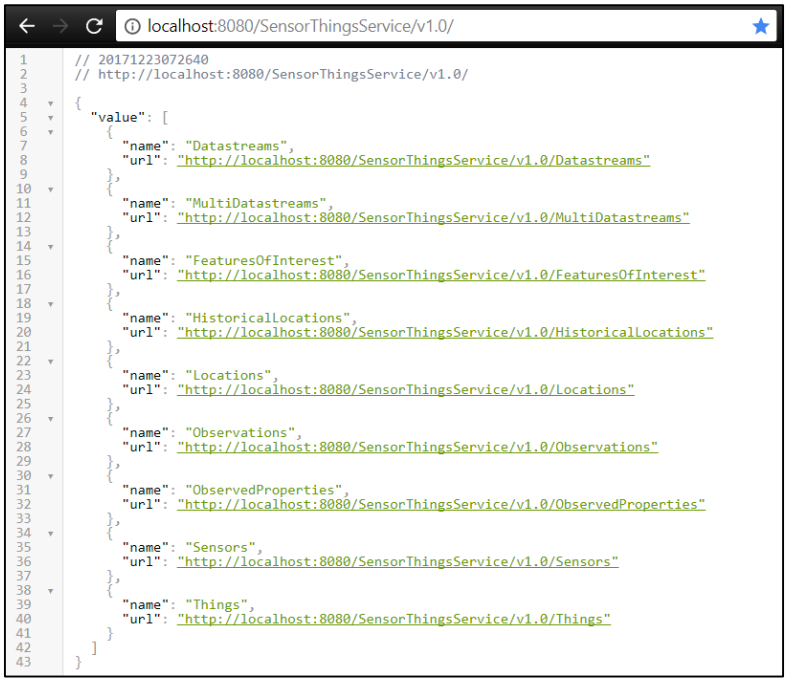
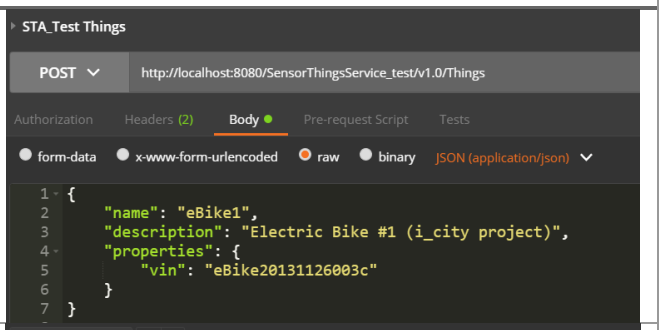
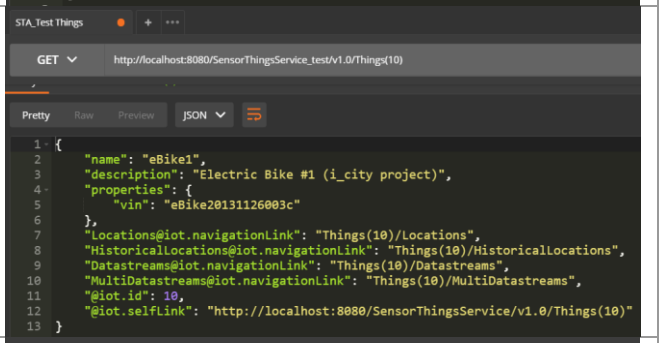
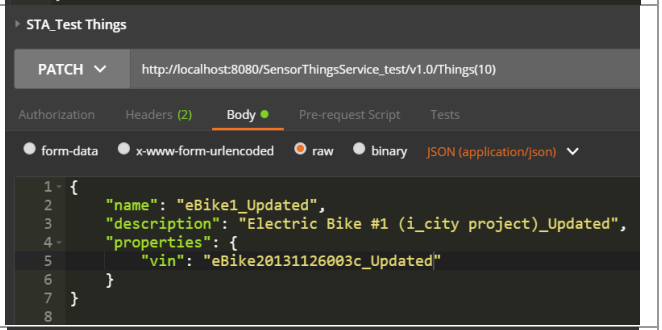
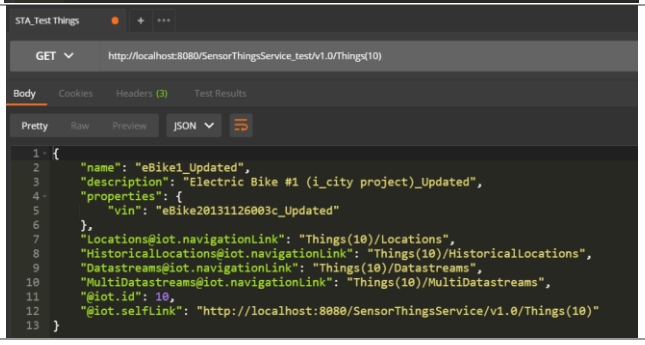
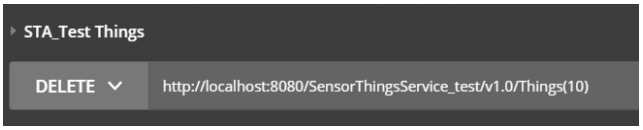
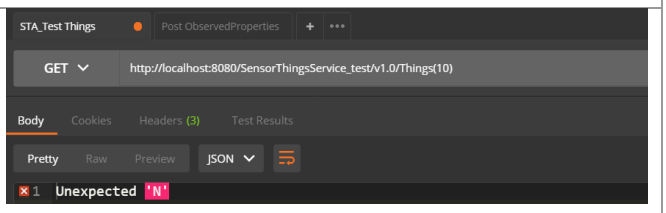


Figure 25: SensorThings API base resource path ("http://localhost:8080/SensorThingsService/1.0")

The next step is to check if all functionalities of RESTful HTTP service are working well with all entities. To do this, the application "Postman" is used to check all entity links, as it is the most-used REST client worldwide and have an intuitive user interface to send requests, save responses, add tests, and create workflows. [77] The test of the local SensorThings server implementation is done by making a "POST" request to send some data to the entity database and then making a "GET" request to receive that data following by making a "PATCH" request to update the data and finally making a "DELETE" request to delete that data from database. The request commands and results are shown in (Table 3).

Table 3: Test the SensorThings API requests and responses

Request description	screenshot from Postman request
1. Making POST request to add "Thing" entity to the SensorThings Server. Request URL: http://localhost:8080/SensorThingsService_test/v1.0/Things Request body: {"name": "E-bike1_Update","description": "Electric Bike #1 (i_city project)","properties": {"vin": "E-bike20131126003c"} } Response body: -	
2. Making GET request to check the response body if the latest Thing entity has already updated by POST request. Request URL: http://localhost:8080/SensorThingsService_test/v1.0/Things(10) Request body: - Response body: Shown on the right screenshot. (Showing that latest Thing ID =10)	
3. Making PATCH request to change some value of the Thing entity. (Thing ID = 10) Request URL: http://localhost:8080/SensorThingsService_test/v1.0/Things(10) Request body: {"name": "E-bike1_Update","description": "Electric Bike #1 (i_city project)","properties": {"vin": "E-bike20131126003c"}} Response body: -	
4. Making GET request to check the response body if the latest Thing entity has already updated by PATCH request. (Thing ID = 10) Request URL: http://localhost:8080/SensorThingsService_test/v1.0/Things(10) Request body: - Response body: Shown on the right screenshot.	

Request description	screenshot from Postman request
5. Making DELETE request to delete the latest Thing entity. (Thing ID = 10) Request URL: http://localhost:8080/SensorThingsService_test/v1.0/Things(10) Request body: - & Response body: -	
6. Making GET request to check if the response body is empty to confirm that the Thing has already deleted from the SensorThings server. Request URL: http://localhost:8080/SensorThingsService_test/v1.0/Things(10) Request body: - & Response body: -	

4.2.2 OGC Sensor Observation Service (SOS)

To install the OGC Sensor Observation Service in this research, the implement version of SOS from 52°North company [78] is used and it is called "52°North Sensor Observation Service (SOS)". The latest version of 4.4 is used and downloaded from 52°North Github. [14] From this source, the web app has already built in WAR format and ready to be deployed in Apache Tomcat server. The basic architecture of the 52°North SOS 4.4 is shown in (Figure 26). The database that connected to the SOS is a PostgreSQL database with PostGIS extension.

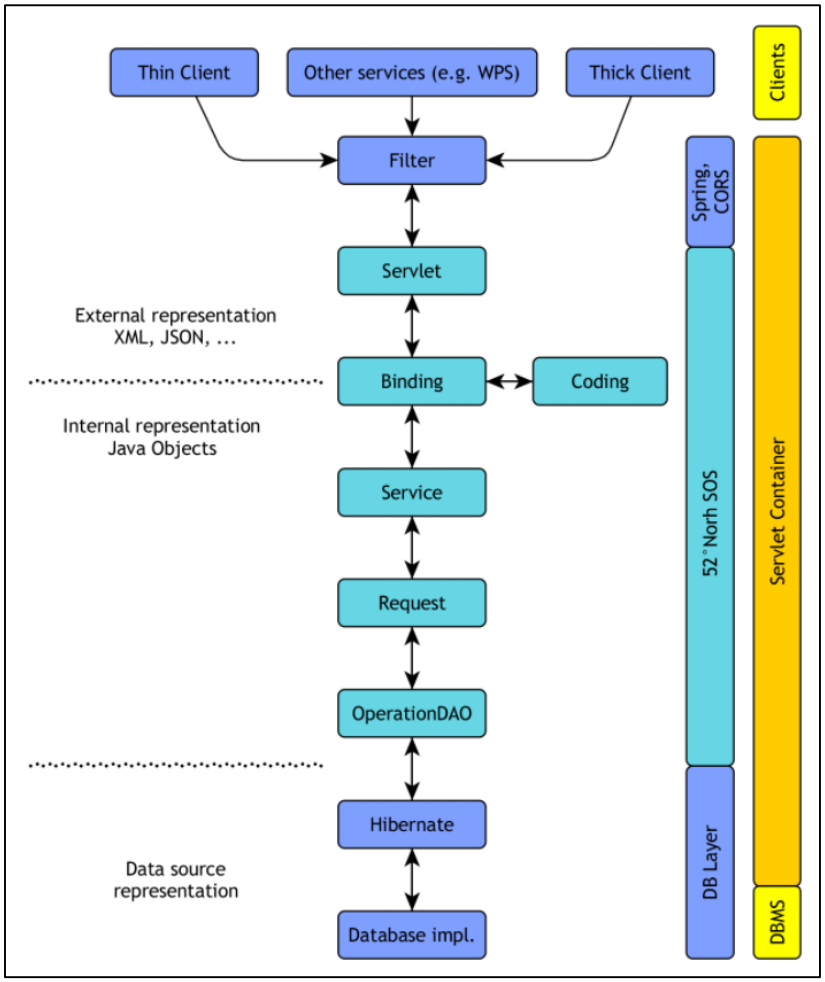


Figure 26: basic architecture of the 52°North SOS 4.4

After the installation of the 52°North SOS 4.4 is done, in this version the 52°North also provides the web application to make the first configuration to administrate the basic setting for SOS such as a database connection, a minimum, and a maximum number of connection pool size, batch size, etc. Then, the transactional SOS operation test is made through the 52°North SOS Test Client. (Figure 27) Before the SOS can be tested, all "Insert" operations have to be activated which is normally inactivated by default. (Figure 28)

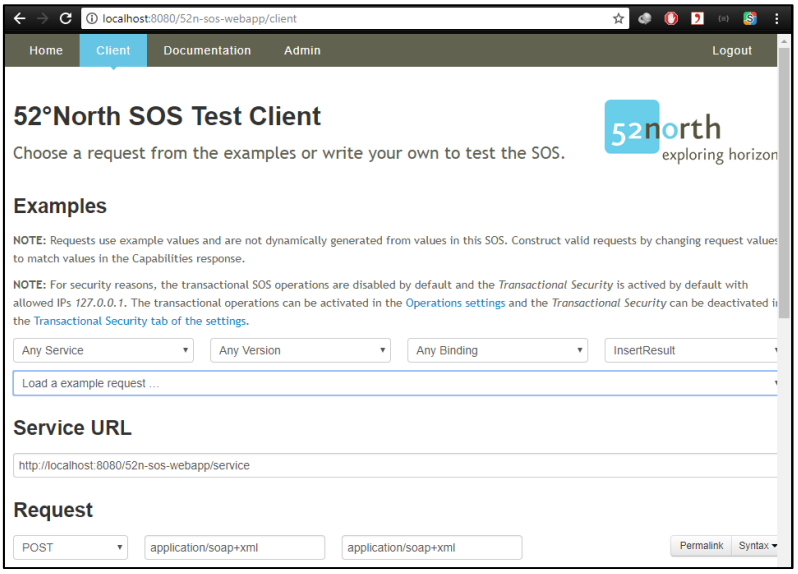


Figure 27: 52°North SOS Test Client Web Application

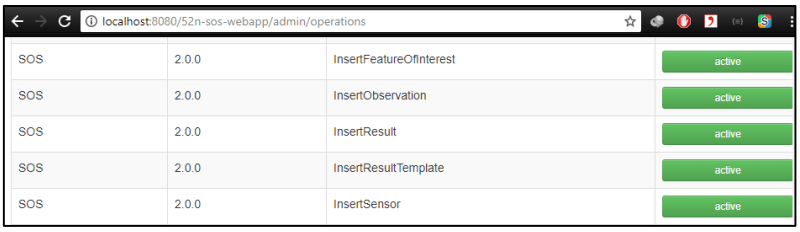


Figure 28: 52°North SOS admin operation activation

With the 52°North SOS Test Client application all operation methods can be tested through the web browser. Beginning with inserting new sensor into the server by "InsertSensor" operation which the "Sensor" observation results can only be inserted for sensors that have already been inserted in the SOS. [42, 79] Although this version of SOS is supported to the JSON format binding at the endpoint, still, part of the request and response bodys is an XML based description. To test the "InsertSensor" operation,a test is done with an example request dataset and the operation request is attached in (Appendix B - 1 : 52°North Sensor Observation Service "InsertSensor" operation). Then, the operation to insert result template is done to prepare the template for inserting the result value. To add this part, the "observationTemplate" has to be specified followed by the OM_Observation specification in [80]. The operation request test for this part is attached in (Appendix B - 2 : 52° North Sensor Observation Service "InsertResultTemplate" operation The response to this request will show the templateIdentifier which is used as a reference later on to add the result to the server. Then, after the Sensor and ResultTemplate have been specified in the SOS, the result can be imported into the server by using "InsertResult" operation. The operation request for this part is attached in (Appendix B - 3 : 52° North Sensor Observation Service "InsertResult" operation request).

4.2.3 Advanced Sensor Data Delivery Service (ASDDS)

To implement the Advanced Sensor Data Delivery Service (ASDDS), the Ubuntu 17.10 operate system [81] is used and installed in Oracle VM VirtualBox [82] virtualization product. Firstly, the database in PostgreSQL is installed followed by editing a configuration of yml file to match the database connection including database name, username and password. Then, the database schemas are built up in the first set-up by using the command shown in (Figure 29). After that, the server can be started with the command in (Figure 30). Then, after the server start, the terminal screen will show up as in (Figure 31).

```
“java -jar GeoVisServer-0.0.1.jar db migrate config.yml”
```

Figure 29 : a Java command to build up the database schemas

```
"java -jar GeoVisServer-0.0.1.jar server config.yml"
```

Figure 30: a Java command to start up the ASDDS server.

Figure 31: Terminal screen when the ASDDS server start

After the server is running, the configuration in the python program “client.py” is made to match the URL name, port, username and password. Then, the observation data can be imported into the ASDDS server with the following command in (Figure 32). Next, the imported observation log will be shown up in the terminal screen as in (Figure 33).

```
Python3 client.py (filename)
```

Figure 32: Python command to import observation data

```
joe@joe-VirtualBox:~/Desktop/ASDDS/GeoVisIDataImporter$ python3 client.py a.csv

 ------------------------------------------------------------
 Read File
 ------------------------------------------------------------


 ------------------------------------------------------------
 Insert Sensors and Observation's
 ------------------------------------------------------------

Add Sensor: [Name: DEBW522AA00000b82 Description: DEBW522AA00000b82 Unit: kWh Ad
dress: DEBW522AA00000b80 SerieId: 0]
Add Observation: [Value: 2 Timestamp: 2014-01-21 20:00:00]
Add Observation: [Value: 3 Timestamp: 2014-02-11 16:00:00]
Add Observation: [Value: 3 Timestamp: 2014-03-04 12:00:00]
Add Observation: [Value: 2 Timestamp: 2014-03-25 08:00:00]
Add Observation: [Value: 1 Timestamp: 2014-04-15 05:00:00]
Add Observation: [Value: 0 Timestamp: 2014-05-06 01:00:00]
Add Observation: [Value: 0 Timestamp: 2014-05-26 21:00:00]
```

Figure 33: Terminal screen showing the imported observation to the server

Next, the request for the observation results can be tested in the web application made with "Swagger" [83]. After that, the results from ASDDS will be returned in JSON format with the fixed structure containing objects of "value", "timestamp", "sensorName" and "timestampAsString". The object "location" can also be identified or added to the ASDDS but only with the string value available.The response body of the tested observation results are shown in (Figure 34).

Request URL

```
http://localhost:8888/api/observation
```

Response Body


```
[
  {
    "value": 1,
    "timestamp": 1388530800000,
    "sensorName": "DEBW522AA00000178",
    "timestampAsString": "2014-01-01T00:00:00.000+01:00"
  },
  {
    "value": 1,
    "timestamp": 1388534400000,
    "sensorName": "DEBW522AA00000178",
    "timestampAsString": "2014-01-01T01:00:00.000+01:00"
  },
  {
    "value": 1,
    "timestamp": 1388538000000,
    "sensorName": "DEBW522AA00000178",
    "timestampAsString": "2014-01-01T02:00:00.000+01:00"
  },
```


Figure 34: Response body of the tested observation results from ASDDS server through the Swagger.

4.3 Sensor Data Management

In this research, the sensors or IoT devices are including four TCU-sensors from four Smart Electric Bikes, one Garmin Smart Watch, and the open weather data of the Stuttgart city. The datastreams from these sensors have to be prepared in order to make them synchronized with the sensor network.

4.3.1 Network Architecture

In order to manage the data from heterogeneous sensor systems, firstly, the clear architecture structure must be clarified. The network architecture of this research from each device to the user application level can be concluded as in (Figure 35).

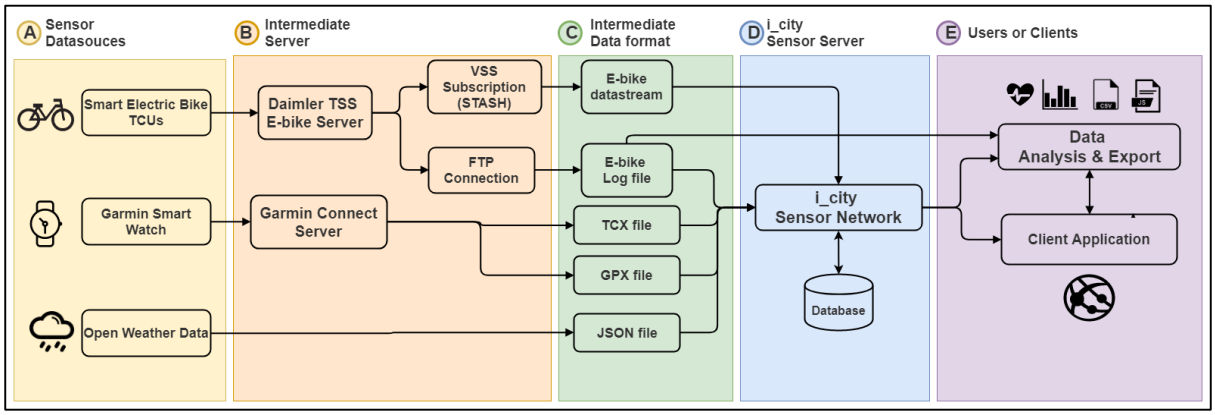


Figure 35: Overall Network Architecture

The first part "Ⓐ Sensor Data sources" shows the sensor data sources of this work which contains three main data sources including "Smart Electric Bike TCUs", "Garmin Smart Watch" and "Open Weather Data". Following by part "Ⓑ Intermediate Server" which shows the intermediate servers of the Smart E-bike TCU and Garmin Smart Watch data. For the E-bike TCU server, to set up the sensor network for receive the E-bike data in the local server from the existing architecture in i_city project (Chapter 3.3), the first solution is to deploy the sensor network called "STASH" which will subscribe the E-bike data from VSS server directly into the local server. It is an implementation of the E-Bike data architecture from the research of Daimler TSS. [4] With this solution, the E-bike data are sent as the data streams to the server whenever there is update data logs on the E-bike sensors. However, due to the network security issue, there is a delay to complete this network deployment. So that, the second solution called FTP solution which the communication from the HFT and Daimler TSS are made via FTP method. With this solution, users can retrieve all the data with the update every 24 hours at 02:00 am. The background of this method is that the E-Bike TCU datastreams are collected into with the database of VSS server then the data are upload into the FTP server in the period of 24 hours. With this method, the data are sent as log files containing many data streams which can compensate the setup delay of the first solution. However, the drawback of this solution is that the i_city researchers cannot get the data in real-time and have to wait until 02:00 on the next day to get the data on each day. For the Garmin Smart Watch data, the datastreams are sent to the Garmin Connect server [84] when the device is connected to the internet. Then, the data can be extracted into TCX or GPX format later on from the web application interface. This part is managed manually by Jan Eric Silberer (Chapter 3.2). In the future work, the real-time connection from each sensor device could be developed.

Then, the next part "© Intermediate Data Format" shows the different data formats from each sensor system. For the E-bike data, the data are divided into two groups according to the server part which this research is mainly focusing on the E-bike log file because of the limitation on the usage of E-bike datastreams from VSS subscription. As the other research was going on while the i_city sensor network on the HFT is still under developed, so that the E-bike log files are converted into the table format directly to the users in part "Ⓔ Users or Clients" directly and also imported into the i_city sensor network in part "Ⓓ i_city sensor server" as well. So that the data conversions are needed in both ways. For the data from Garmin Smart Watch, the data are provided in formats of GPX and TCX but some fields or entities exists uniquely only in one format so that both formats are needed to maintain all the valuable data without losing any data. For the data from OpenWeatherData, the data is already in JSON format which is not needed for cleaning or conversion. The more details on data conversion of the E-bike and Garmin Smart Watch data are discussed in (Chapter 4.3.2).

The part "Ⓓ i_city sensor server" is the implementation of the sensor networks in this research including (1) OGC SensorThings API, (2) OGC SOS and (3) Advanced Sensor Data Delivery Service (ASDDS) which the details are shown in (Chapter 4.2). Following by the last part "Ⓔ Users or Clients" which is the part for mainly utilizing the data from the implemented sensor network and the details are shown in (Chapter 4.5).

4.3.2 Data Cleansing

As stated in the last chapter about the network architecture (Chapter 4.3.1), the data cleansing is needed from the section "© Intermediate Data Format" to "Ⓓ i_city sensor server" or "Ⓔ Users or Clients". To specify in details, the following figure (Figure 36) shows the network architecture with the symbols from "C1" to "C6" on each stage.

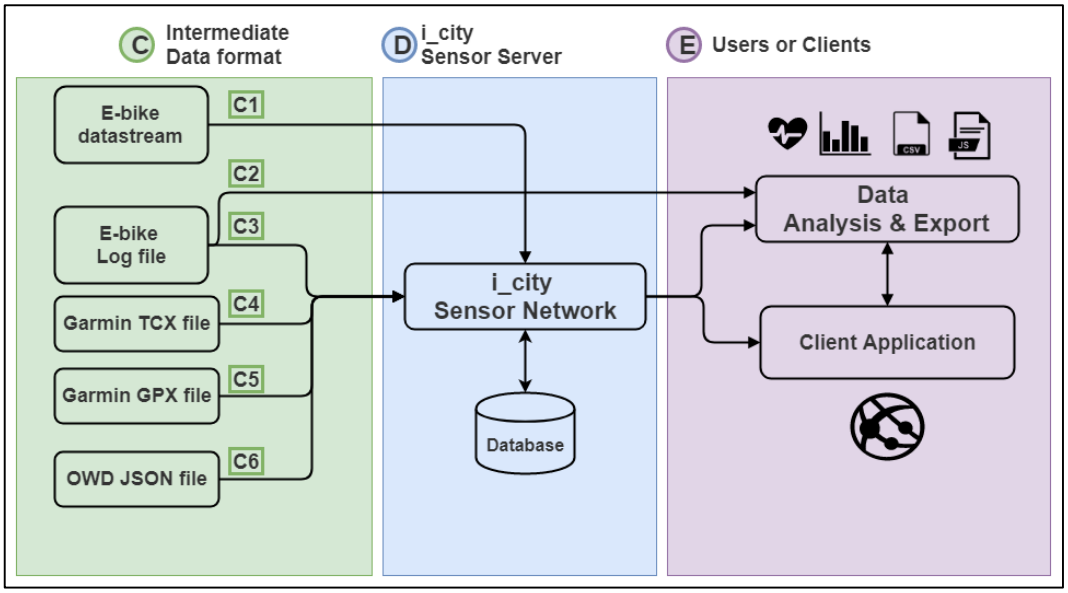


Figure 36: Network Architecture focusing on the Intermediate Data format to i_city Sensor Network or Users or Clients

For the E-bike data which are divided into two groups, due to the limitation on usage of E-bike datastreams from VSS subscription so this research will use the data of E-bike log file (Figure 36 – C2 and C3). To specify in more details on these FTP log files, they contain the E-bike data from all E-bike TCUs in the systems and each line of data is collecting the data as a JSON object, however, those JSON objects are not built as the JSON arrays as they should be, so these files are not valid as a JSON. The example of the E-bike log events retrieve by FTP solution is shown in (Appendix A - 4 : Example of the Smart E-bike log data retrieve by FTP solution). So that the program to automatically change these file into valid JSON array is needed.

After the files are valid JSON format, the data cleansing is needed. As one part of the JSON body provides the latest basic status of E-bike such as “fuelLevel”, “charging”, “geolocation”, “powerState”, “light”, “VIN (Vehicle Identification Number)”, “batteryVoltagE-bike”, “mileage”, “batteryLevel”, and “TimeStamp” which are always existing. Another part of the JSON body provides the latest update data of E-bike such as “speed” and “pedalForce” which are existing only when there is an update of the data.

The full lists of the JSON object body structures are shown in (Appendix A - 3 : Smart Electric Bike JSON API response body structure). For example, the E-bike JSON body messages from E-bike (VIN id: E-bike20131127000a) at 2017-11-21_02:07:28.162 are shown in (Figure 37). It can be seen that there is some duplicated data field such as “VIN” and “batteryVoltagE-bike” and some missing field such as “speed”. According to this issue, the program to determine the existence of the data before importing into the sensor network is written and explained in (Chapter 4.3.4).

```
{
        "change": {
            "batteryVoltageBike": 53351
        },
        "vin": "eBike20131127000a",
        "uuid": "8732be68-2d66-4665-b21b-b90b5ad35ffa",
        "status": {
            "allDoorsClosed": true,
            "fuelLevel": 100,
            "charging": true,
            "ignitionOn": false,
            "geo": {
                "latitude": 48.42233424,
                "accuracy": 4,
                "longitude": 9.94182737
            },
            "powerState": "ON",
            "light": false,
            "connection": {
                "connected": true,
                "since": "Nov 21, 2017 12:07:30 AM"
            },
            "vin": "eBike20131127000a",
            "locked": false,
            "batteryVoltageBike": 53351,
            "mileage": 4092,
            "batteryLevel": 5.403541
        },
        "timestamp": "2017-11-21_02:07:28.162"
}
```


Figure 37: the E-bike JSON body message from E-bike (VIN id: E-bike20131127000a) at 2017-11-21_02:07:28.162

Moreover, as stated in Chapter 4.3.1, the other research are depending on this data log while the i_city sensor network is still under developed so the data conversion into general format as "CSV" or "XLS" is needed to make this E-bike log data be able to analyzed by other researchers in i_city project in a convenient way. This conversion is related to the architecture in (Figure 36 – C2). There are some plenty tools to convert the JSON file into the CSV format such as a Microsoft Excel or an online converter such as "JSON to CSV Converter" [85]. However, the input log file is not valid as a right JSON format. So that, the program to adjust this file to convert it into a JSON array and convert it into a CSV is written and is shown in (Appendix A - 5 : NodeJS program for converting the Smart E-bike log data into CSV). This program is written in JavaScript and use NodeJS as a JavaScript runtime and use "json2csv" JavaScript package [86] to converts the JSON array into CSV file format with specified column titles. The program is easy to use as users can put the file path to the E-bike log file (.log) into the program and run the program, then the program will automatically generate the output table file in CSV format with the same root name of the input file. The example of the output CSV file is shown in (Appendix A - 6 : Example of the Smart E-bike log data after converting into CSV format). From this output data, the researchers in the i_city project can use it as an input file to do the data analysis in a spreadsheet analysis software such as Microsoft Excel and SPSS. In the same way, these E-bike data logs are also imported into the sensor network directly (Figure 36 – C3)

The next part is the Garmin Smart Watch data, as in this research the Garmin Smart Watch fēnix® 5X [51] is used as the sensor for collecting the health data of the E-bike user. The raw files provided by Garmin Smart Watch fēnix® 5X are including "TCX" (Figure 36 – C4), "GPX" (Figure 36 – C5) and "KML" format. The TCX or Training Center XML format is a data exchange format introduced in 2007 as part of Garmin's Training Center product which is in XML base. It provides the important data field including "Timestamp", "Total Distance (m)", "Heart Rate (bpm)", "Speed (km/h)" and user coordinate (Latitude (°) / Longitude (°) / Altitude (m)). While the GPX or GPS Exchange Format contains almost all same important field of data as in the TCX file including the "Temperature" data which is not provided in TCX file, but the data of "Total Distance (m)" and "Speed (km/h)" are missing in the GPX file. So that, both files are used to provide the complete data in the database. The real sample collecting data in TCX and GPX are showing in (Figure 38) and (Figure 39) respectively. The more example of the observation data is shown in (Appendix A - 1 : The example of the Observation data from Garmin Smart Watch fēnix 5X (GPX-file) collected on 2017-11-21, Appendix A - 2 : The example of the Observation data from Garmin Smart Watch fēnix 5X (TCX-file) collected on 2017-11-21).

```
<Trackpoint>
            <Time>2017-11-21T14:40:33.000Z</Time>
            <Position>
              <LatitudeDegrees>48.78198646008968</LatitudeDegrees>
              <LongitudeDegrees>9.177944166585803</LongitudeDegrees>
            </Position>
            <AltitudeMeters>212.8000030517578</AltitudeMeters>
            <DistanceMeters>40.220001220703125</DistanceMeters>
            <HeartRateBpm>
              <Value>75</Value>
            </HeartRateBpm>
            <Extensions>
              <ns3:TPX>
                <ns3:Speed>0.7649999856948853</ns3:Speed>
              </ns3:TPX>
            </Extensions>
          </Trackpoint>
```

Figure 38: Sample TCX data collected by Garmin Smart Watch fēnix® 5X at 2017-11-21T14:40:33.000Z

```
<trkpt lat="48.781986460089683532714843750" lon="9.177944166585803031921386718750">
        <ele>212.8000030517578125</ele>
        <time>2017-11-21T14:40:33.000Z</time>
        <extensions>
          <ns3:TrackPointExtension>
            <ns3:atemp>32.0</ns3:atemp>
            <ns3:hr>75</ns3:hr>
          </ns3:TrackPointExtension>
        </extensions>
      </trkpt>
```

Figure 39: Sample GPX data collected by Garmin Smart Watch fēnix® 5X at 2017-11-21T14:40:33.000Z

The last data source for this research is the historical weather data from OpenWeatherMap [12], it is an IT company provide a global geospatial platform which is affordable to users and enables

them to freely operate with Earth Observation data especially in this study the weather data. The hourly historical weather data in the city of Stuttgart at the station location of 48°46'56.3"N 9°10'37.3"E is used. [87] The API response from this data source is returned in JSON format. The following figure (Figure 40) shows the example of the JSON response for the weather data in Stuttgart in 1st October 2017 at 0:00:00. As this data is already in JSON format it is straightforward to be imported into the SensorThings API server. A full document of all parameter details is attached in (Appendix A - 7 : OpenWeatherMap Weather parameters in API respond for hourly historical data for cities [87]).


```
{
    "city_id": 2825297,
    "main": {
        "temp": 284.15,
        "temp_min": 284.15,
        "temp_max": 284.15,
        "pressure": 1020,
        "humidity": 100
    },
    "wind": {
        "speed": 2,
        "deg": 250
    },
    "clouds": {
        "all": 75
    },
    "weather": [
        {
            "id": 803,
            "main": "Clouds",
            "description": "broken clouds",
            "icon": "04n"
        }
    ],
    "dt": 1506816000,
    "dt_iso": "2017-10-01 00:00:00 +0000 UTC"
}
```


Figure 40: JSON response for the weather data in Stuttgart in 1st October 2017 at 0:00:00

4.3.3 *Sensor Data Modelling*

In this research, the sensor protocol conceptual model is based on the OGC Observations and Measurements (O&M) specification [21]. So that, firstly, all the "Sensing" entities have to be all specified starting with modeling the incoming E-bike data in the i_city project, the "Things" entity set is referred to the E-bikes while the "Sensors" entity set is referred to sensors equipped on each E-bikes. For the "ObservedProperties" entity set, it is referred to the data type of the observed value from E-bike TCU. Then, the "FeaturesOfInterest" entity set is referred to users or riders. As there is no system to automatically identify who uses the E-bikes at the moment, so in a current version of the sensor network, the "FeaturesOfInterest" entity is set to the E-bike's user ID manually which there is a list of user information linked to this ID managed by Jan Eric Silberer. For the location of the E-bikes, the

latest updated locations will be collected and updated in the "Location" entity and the historical data will be automatically moved to the "HistoricalLocation" entity whenever there is a new update data of the location. So that the developer can access to all E-bike location data in both real-time and historical dataset. The example data of the mentioned entities are shown in (Table 4).

Table 4: Example data from Observations and Measurements entities. (Things, Sensors, ObservedProperties, FeaturesOfInterest, Location and HistoricalLocation entities)

Entity	Refer to ...	Count	Example data
Things	E-bike data and property	4	**"name"**: "E-bike1", **"description"**: "Electric Bike #1 in i_city project", **"properties"**: *{"vin": "E-bike20131126003c"}*
Sensors	E-bike TCU data and property	4	**"name"**: "E-bike1-TCU", **"description"**: "Telematic & Connectivity Unit ", **"encodingType"**: "application/pdf", **"metadata"**: "https://www.smart.com/....pdf"
ObservedProperties	E-bike Observed data	20	**"name"**: "eb.fuelLevel", **"description"**: "The incoming E-bike data indicates the amount of remaining bicycle battery provided in % "
FeaturesOfInterest	E-bike user data	1	**"name"**: "E-bike Users", **"description"**: "Bike usage data from i_city user", **"encodingType"**: "application/vnd.geo+json", **"feature"**: *{"type": "Point",* *"coordinates": [9.1726,48.7803]}*
Location	The latest location of E-bikes	Increasing overtime	**"name"**: "HFT Stuttgart", **"description"**: "UNIVERSITY OF APPLIED SCIENCES STUTTGART, GERMANY (Starting Point)", **"encodingType"**: "application/vnd.geo+json", **"location"**: *{"type": "Point",* *"coordinates": [9.1726, 48.7803]}*
HistoricalLocation	The historical location of E-bikes	Increasing overtime	**"name"**: "HFT Stuttgart", **"description"**: "UNIVERSITY OF APPLIED SCIENCES STUTTGART, GERMANY (Starting Point)", **"encodingType"**: "application/vnd.geo+json", **"location"**: *{"type": "Point",* *"coordinates": [9.1726, 48.7803]}*

Then, the "Datastreams" entity set is referred to the unique ObservedProperty, Sensor, Thing and FeatureOfInterest entity set. To confirm the idea about "Datastreams", the example figure explains each entity in Observations and Measurements concept linking to the E-bike usage in an i_city project showing in (Figure 41). As a result, the number of all possible Datastreams is equal to the multiplication of these four entities which should be [*4 Things x 4 Sensors x 20 ObservedProperties x1 FeaturesOfInterest = 320 Datastreams*]. However, in this research, each TCU is set to the particular E-bike which means the relation of Things and Sensors entity is set as one to one, so the number of all possible "Datastreams" is [*4 (Sensors and Things) x 20 ObservedProperties x1 FeaturesOfInterest = 80 Datastreams*]. The example data of the mentioned entities are shown in the following table. (Table 5) The full lists of all

"Datastreams" entity sets are shown in (Appendix B - 9 : Relation table of the "Datastreams" entity to "ObservedProperties", "Sensors" and "Things" entities in SensorThings API server).

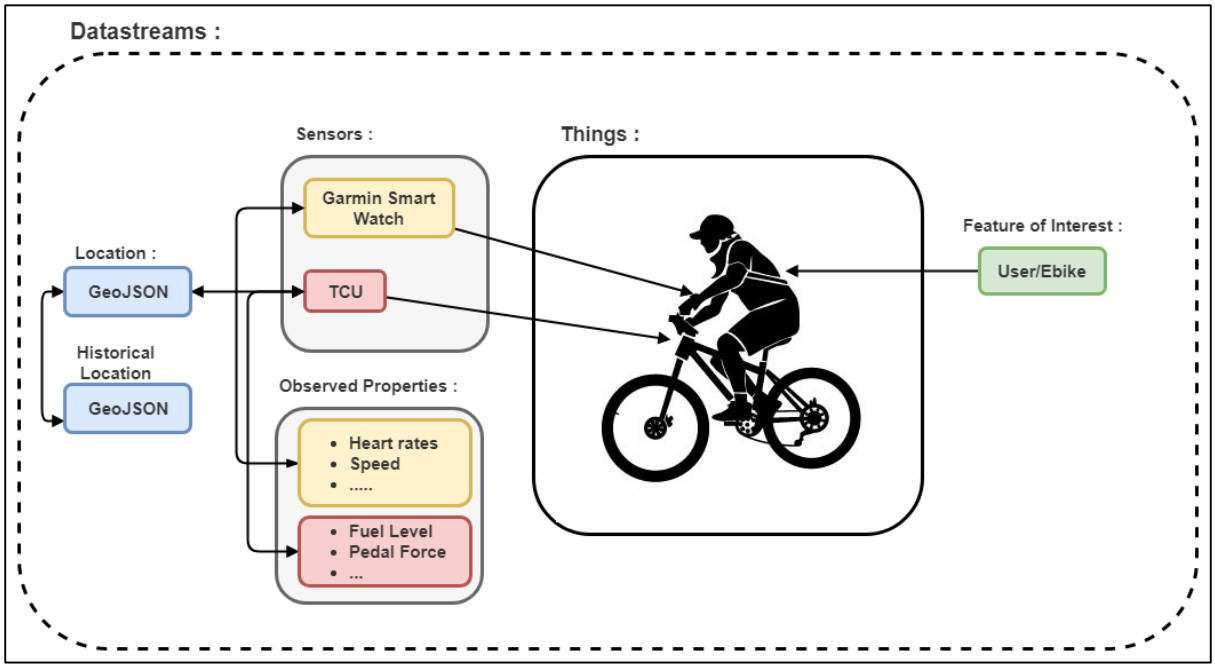


Figure 41: The example of the Observations and Measurements entities for E-bike usage in i_city project (Bike image from [88])

Table 5: Example data from Observations and Measurements entities. (Datastream and Sensors entities)

Entity	Refer to ...	Count	Example data
Datastream	A unique ObservedProperty, Sensor, Thing and FeatureOfInterest	80	**"name"**: "E-bike1", **"description"**: "Electric Bike #1 in i_city project", **"properties"**: *{"vin": "E-bike20131126003c"}*
Sensors	E-bike TCU data and property	4	**"name"**: "E-bike1-TCU", **"description"**: "Telematic & Connectivity Unit ", **"encodingType"**: "application/pdf", **"metadata"**: "https://www.smart.com/....pdf"

After all the entities of Observations and Measurements specification are identified to use in this research, the relations of entities are constructed based on the O&M standard which is described in (section 3.1.1 : OGC SensorThings API). As a result, (Figure 42) are showing the UML diagram of SensorThings API for this research with the example incoming data indicating the fuel level of E-bike which has VIN "E-bike20131126003c".

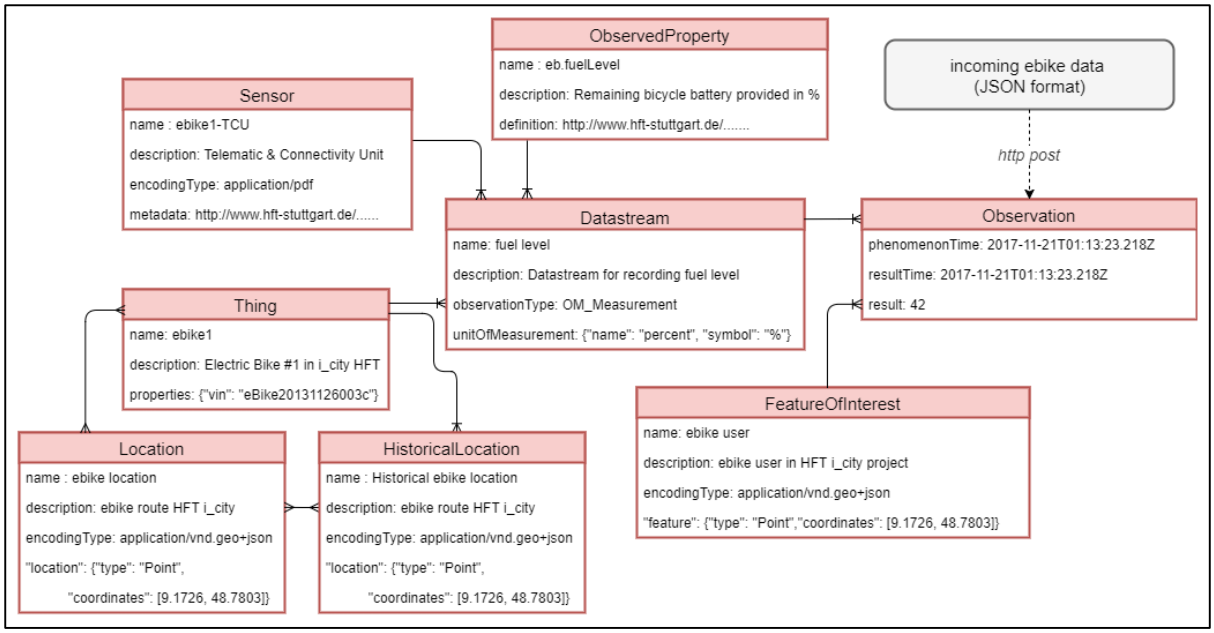


Figure 42: UML diagram showing example for storing E-bike data of the datastream that collecting fuel level of the E-bike1 (VIN: E-bike20131126003c) based on SensorThings API Standard

To use SensorThings API to manage the incoming data as an Observation entity, all possible "Datastream" and "FeatureOfInterest" must be all identified, created and imported to the sensor network. As the "Datastream" entity is a set of combination of "Sensor", "Thing", "ObservedProperty" and "Location" entities, so firstly these entities must be identified. To do this, the JSON arrays of these entities are created followed by the SensorThings API standard. For example, the following figures show some part of the JSON array of the "Things" entities in (Figure 43) and the "Sensors" entity in (Figure 44). The full lists of JSON array of "Sensors" are shown in (Appendix B - 4 : The initial values of the "Sensors" entity in JSON file for importing to SensorThings API server). The full lists of JSON array of "Things" are shown in (Appendix B - 5 : The example initial values of the "Things" entity in JSON file for importing to SensorThings API server). The full lists of JSON array of "ObservedProperties" of the Smart Electric Bike are shown in (Appendix B - 6 : The example of the initial value of the "ObservedProperties" entity of the Smart Electric Bike in JSON file for importing to SensorThings API server). The full lists of JSON array of "ObservedProperties" of the Garmin Smart Watch are shown in (Appendix B - 7 : The example of the initial values of the "ObservedProperties" entity of the Garmin Smart Watch in JSON file for importing to SensorThings API server). For the initial value for "Location" entity, the location of HFT Stuttgart is used as the starting point for all sensors and devices in this research. The JSON describing the "Location" entity is shown in (Appendix B - 8 : The initial value of the "Locations" entity in JSON file for importing to SensorThings API server).

```
[
{
    "name": "eBike1",
    "description": "Electric Bike #1 (i_city project)",
    "properties": {
        "vin": "eBike20131126003c"
    }
},
...
]
```

Figure 43: An example of JSON array showing the first "Things" entity.

```
[
  {
    "name": "eBike1-TCU",
    "description": "Telematic & Connectivity Unit sending ebike data",
    "encodingType": "application/pdf",
    "metadata": "https://www.smart.com/content/dam/smart/EN/PDF/smart_ebikeFlyer_2013_E-
int.pdf"
  },
...
]
```

Figure 44: An example of JSON array showing the first "Sensors" entity.

After the JSON arrays of those entities are prepared and ready, they are added to the SensorThings server by POST requests which the step can be described and concluded in (Figure 45). All entities including "Things", "Sensors", "Locations", "FeaturesOfInterest" and "ObservedProperties" are posted into the sensor server and then the "Datastreams" entity can be built based on the other entities in the server. Then, the sensor network is ready to import the "Observations" entity or the observation data from each sensor which is described in details in (Chapter 4.3.4 : Importing the Data into the Sensor Network).

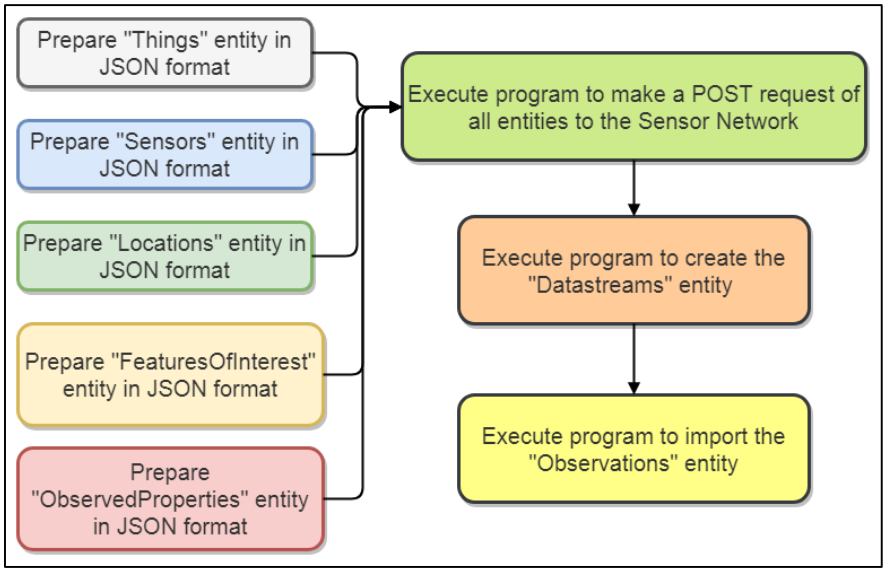


Figure 45: The steps for managing the Sensor Data Modelling

To prepare all the Sensing entities, the HTTP request can be done through POSTMAN application, but as the limitation of the SensorThings API that it is supported to receive a request with only a single JSON object each time, for example, making POST requests of four "Thing" entities need four times POST requests. So that, the automatic program to make a POST request of the entire JSON arrays is created for this research and it can be used in the other projects as well. This program is written in JavaScript based and needs NodeJS as a JavaScript runtime.Later on, this program can be used to do the post request on the server-side in the i_city project. The two main NodeJS packages used in this program are "jsonfile" package for easily utilize the JSON file and "request" package for creating a POST request. [76, 89]. The automatic program for automatically making "Post" request method to SensorThings API Server of all SensorThings API entities is written and a full source code of this automatic program will be attached in Appendix Disc. (Appendix C)

4.3.4 Importing the Data into the Sensor Network

For the incoming E-bike observation data, it is sent as a log data which is stored as a JSON object for each line but not valid as a JSON file as explained in (Chapter 4.3.2 : Data Cleansing). So that, before this data can be imported to SensorThings API, it must be adjusted into a single JSON array. This step can be done with NodeJS program by using the "fs" JavaScript package to load the log data and use "replace" function to replace all newline values [\n] with "," and then add "[" to the beginning and "]" to the end of the log file.And then use the function "JSON.parse()" to turn this log data into the JSON array. Then, the next step is to generate the JSON request for each object from the JSON array log data. For the pattern of this log data receiving from E-bike, some data field will only exist if there is a new updated value. So, the conditional programming is written to check which field is contained in each JSON object before creating a POST request to the SensorThings API service. To do this, the right Datastream entity ID has to be identified correctly, this step can be done manually by identifying it in the source code. To simplify this step for the future research to add more SensorThings API entities in the project, a NodeJS function "CheckDSid" is written to check the input of "ObservedProperties" entity name and "Thing" id, and return the callback function value of the correct "Datastreams" ID. Inside this function, there is an input data dictionary "STA_DataStructure" in CSV format describes the "Datastreams" ID, "Things" ID or "Sensors" ID, "ObservedProperties" name, and some other field.The full list of this data for incoming E-bike log data is shown in (Appendix B - 9 : Relation table of the "Datastreams" entity to "ObservedProperties", "Sensors" and "Things" entities in SensorThings API server). This "CheckDSid" function uses the "csv-parse" module [90] to check the data condition with CSV format. The following code shows the simple JavaScript algorithm to check the "Datastreams" ID of the incoming E-bike log data. The full program for importing the E-bike data in a log file format into the sensor network is attached in the Appendix Disc.

```
CheckDSid: function (check, Thing, callb) {
  var parse = require('csv-parse');
  const fs = require('fs');
  var DataStreamID, out;
  fs.readFile('./Data/STA_initialData/STA_DataStructure.csv', function (err, output) {
    if (err) throw err;
    out = output;
    parse(out, { comment: '#' }, function (err, output) {
      for (let i = 0; i < output.length ; i++) {
        if (check == output[i][1] && Thing == output[i][2]) {
          DataStreamID = output[i][0];
          callb(DataStreamID);
        }
      }
    });
  });
}
```

Figure 46: JavaScript algorithm to check the "Datastreams" ID of the incoming E-bike log data (The full JS program of this function is attached in Appendix Disc)

Then, the next part of this NodeJS program is to loop through all E-bike log data in JSON format which is shown in (Figure 47). For this step, another NodeJS function "generateRequestfromJSON" is written for creating POST request. This loop function is set to run with the "setTimeout" command for limiting the POST request every 100 milliseconds to avoid the problem that NodeJS could loop through the function too fast and generate the non-deterministic order of execution result. The "100 milliseconds" value is chosen by trial and error from [0 to 1000 milliseconds] in the interval of "50 milliseconds" to find the least value that still gives the right execution order results and gives no error during the loop process.

```
function init() {
    for (let i = startLine; i < endLine /*obj2.length*/; i++) {
        setTimeout(function cb() {
            console.log('Read Object : ' + i)
            generateRequestfromJSON(obj2[i], i);
        }, 100 * (i - startLine));
    } // Set the time out dealy of 100 ms
}
```

Figure 47: JavaScript iteration algorithm to create the POST request to the sensor network (The full JS program of this function is attached in Appendix Disc)

In the mentioned loop program, each JSON object of E-bike log data is sent to execute in the function "generateRequestfromJSON" with the input variable "incomingLog" as the log data JSON object and "num" as a number of the object that is sent through this function. The first step in this function is to do the data cleansing by check the "Sensors" entity ID and "Things" entity ID of the log data through function called "executeSTid" if the E-bike VIN number is matched with the E-bike in SensorThings API service as this log data contain the data from all E-bikes apart from this project as well. The function "executeSTid" will return the ID number of the "Sensors" entity or return "0" if the E-bike VIN number is not matched the VIN number in SensorThings API server.

Then, the next step is to adjust the time data value into the ISO 8601 Time string which is an international standard covering the exchange of date- and time-related data, this format is the normal time format to exchange the data among the different sensor devices through SensorThings API server. To simplify this step, the function "replace" from lodash (_) module [91] is used to adjust the time data value in ISO 8601 dates and times format. Then, the conditional function is used to check for the existing field of data, as the incoming data contain only some updated field of data. In the following NodeJS programming code (Figure 48), it shows a function to generate a response body if there is a data field "fuelLevel" from the incoming E-bike log data for making a POST request to SensorThings API server.

```
function generateRequestfromJSON(incomingLog, num) {
    ebike_vin = incomingLog.vin;
    st_id = executeSTid(ebike_vin);
    if (st_id != 0) {
        var dataTime = _.replace(incomingLog.timestamp, '_', 'T');
        if (incomingLog.status.fuelLevel) {
            var id = st_id;
            var dataStr_fuelLevel = {
                "phenomenonTime": dataTime,
                "resultTime": dataTime,
                "result": incomingLog.status.fuelLevel,
                "Datastream": { "@iot.id": id }
            };
            if (execute) {
                postSTA(dataStr_fuelLevel, num);
            } else {
                console.log(`DS fuelLevel: ` + JSON.stringify(dataStr_fuelLevel));
                console.log('----------------------------------------')
            };
        }
        if (…) {
        …
        }
}
```

Figure 48: an example algorithm function to generate the body of the "fuelLevel" Datastream in order to make a POST request into a SensorThings API Server (The full JS program is attached in Appendix Disc)

For the incoming Garmin data, the observation file formats are in TCX and GPX while both of them are the XML based. The reason that both TCX and GPX are both needed to be imported into the server is explained in (Chapter 4.3.2 : Data Cleansing). To manage the data in XML format, the JavaScript package "xml-parse" to parse the XML format as the output JSON is used. [92] Then, they can be imported into the sensor server with the same algorithm to import the E-bike observation data in JSON format. (Figure 48) The full JavaScript program to import the Garmin into sensor server is attached in Appendix Disc. (Appendix C)

For the Open Weather data, the observation file is already in JSON format which the JavaScript program can parse this data and build the datastream body to make a POST request to the sensor server immediately with the same algorithm to import other sensor data. (Figure 48) The full

JavaScript program to import the Open Weather data into sensor server is attached in Appendix Disc. (Appendix C)

4.4 Preparing 3D City Model Data

To do the Web-based Visualization of the 3D City Models on the Cesium's web application with the input CityGML file, the data conversion is needed and can be done with the following solutions including 1) Using 3D CityDB to convert the data into glTF format, 2) Using FME Workbench to convert the data into Cesium 3D Tile format, and 3)Using Georocket GeoToolbox tool to convert the data into Cesium 3D Tile format as shown in (Figure 49).

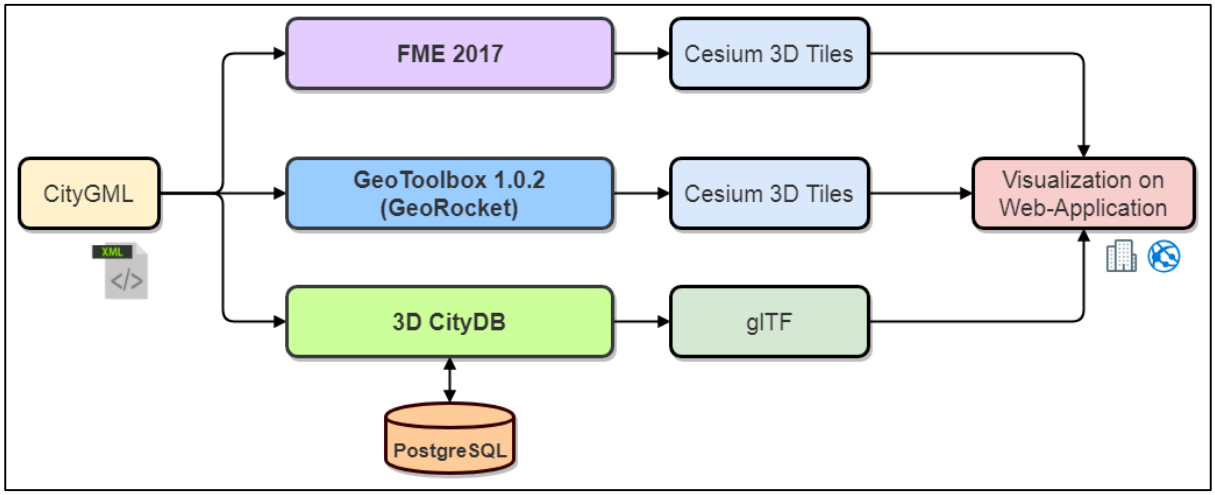


Figure 49: Preparing 3D City Model Data

4.4.1 Using 3D City Database

The "3D City Database" software is a free 3D geodatabase to store, represent, and manage virtual 3D city models on top of a standard spatial relational database PostgreSQL with PostGIS. [69] To implement the 3D City Database to store the CityGML of Stuttgart city area, a full installation of the Importer/Exporter including the 3D City Database SQL scripts, 3D Web Viewer, and documentation is downloaded and installed. [69] Then, the new PostgreSQL database version 9.6 is created with PostGIS extension with the name of "citydb_Hft_Stuttgart_1". After that, the 3D City Database SQL script is used to create an instance of the 3D City Database. The basic connection information is shown in (Figure 50 – left). Then the database schemas are created, the database schema called "citydb" is used to store the 3D data and the database schema called "citydb_pkg" is used to store the functions. (Figure 50 – right)

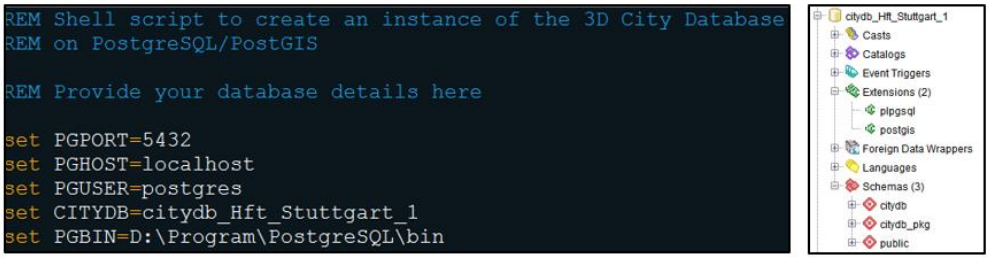


Figure 50: Connection detail of 3D CityDB to PostgreSQL database (left), PostgreSQL database structure after setup the 3D CityDB (right)

After the configuration for the 3D City Database is done, the Stuttgart city CityGML data (Chapter 3.5 : 3D City Model) is imported using the 3D City Database Importer/Exporter [69]. It is a Java-based front-end for the 3D City Database. It allows for reading/writing CityGML instance documents of arbitrary file size. [93] When all the data have been imported successfully, the log messages reporting the imported features into the 3D City Database are shown in the 3D City Database Importer/Exporter console.

```
[01:12:04 INFO] Cleaning temporary cache.
[01:12:04 INFO] Imported CityGML features:
[01:12:04 INFO] IMPLICIT_GEOMETRY: 1
[01:12:04 INFO] APPEARANCE: 421
[01:12:04 INFO] BUILDING: 411
[01:12:04 INFO] BUILDING_PART: 975
[01:12:04 INFO] BUILDING_GROUND_SURFACE: 97
[01:12:04 INFO] BUILDING_ROOF_SURFACE: 6320
[01:12:04 INFO] BUILDING_WALL_SURFACE: 21729
[01:12:04 INFO] GENERIC_CITY_OBJECT: 2
[01:12:04 INFO] SOLITARY_VEGETATION_OBJECT: 1244
[01:12:04 INFO] Processed geometry objects: 89478
[01:12:04 INFO] Total import time: 07 s.
[01:12:04 INFO] Database import successfully finished.
```

Figure 51: the log messages reporting the imported features into the 3D City Database

The next step is to extract the 3D city model data as a glTF format, so the 3D City Database Importer/Exporter [69] is used. It allows 3D city data from the database be exported as glTF models including tiling schemas for visualization in the 3D client application. [93] As the Earth applications running on browsers such as Google Earth, ArcGIS Explorer, Cesium, etc., have decreased their responsiveness greatly with the single large file [94], so tiled exports with small tile file sizes can be used to improve the visualization processing time on the client side. For the example, the hierarchical directory structure for exporting the 2x3 tiles is shown in (Figure 52 - left). For this research, the extracted glTF 3D city model data has the tile structure of 7 tiles (tile 0 - 6) in Latitude direction and 8 tiles (tile 0-7) in Longitude direction. The output directory structure is shown in (Figure 52 - right).

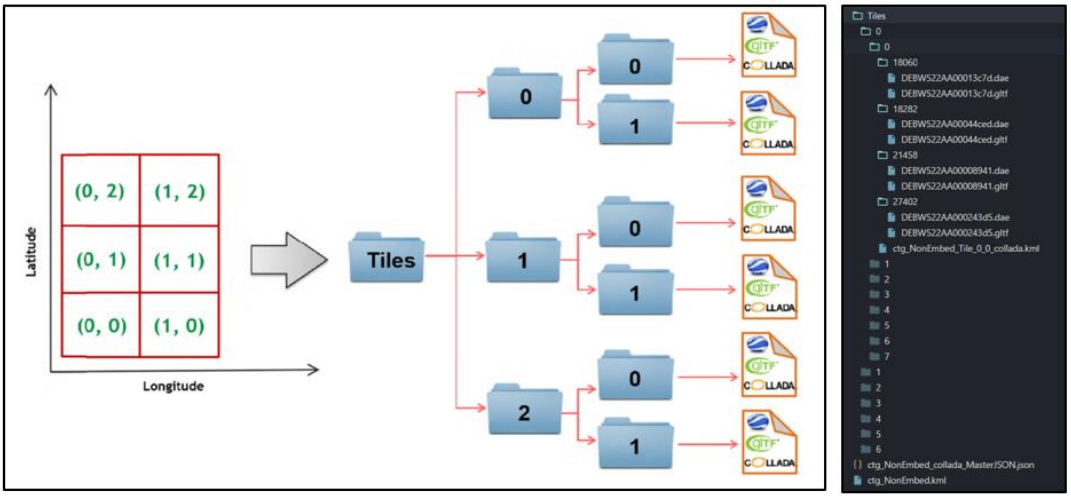


Figure 52: The example hierarchical directory structure for export of 2x3 tiles [93] (left), The output directory structure of glTF tile for Stuttgart 3D city model (right)

To visualize the glTF file on the Cesium Virtual Globe, the JavaScript 3dcitydb-web-map library is used to load the glTF as the 3D City Database layer. This JavaScript library is already included in the full installation of the 3D City Database software package. [69] It is a web-based front-end of the 3DCityDB for high-performance 3D visualization and interactive exploration of arbitrarily large semantic 3D city models in CityGML. [72]

4.4.2 Using GeoRocket GeoToolbox 1.0.2

GeoRocket GeoToolbox is a tool developed by Fraunhofer Institute for Computer Graphics Research IGD. [68] The tool is written in Java and has no user interface. To use this program, all the commands are inserted and run in the Command Window as shown in (Figure 53). The important commands that needed to clarify are shown in (Table 6).

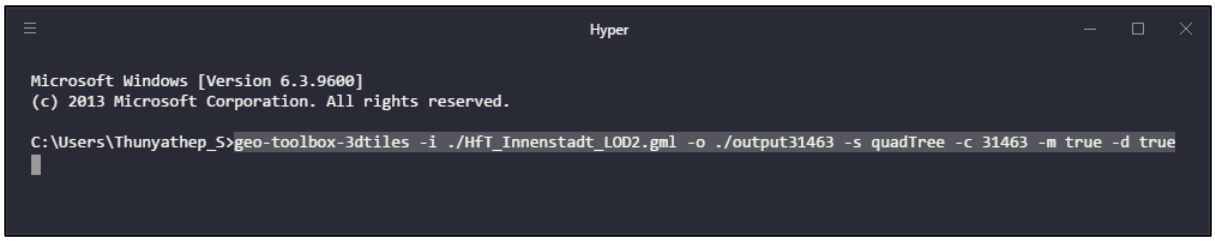


Figure 53: Command to execute the conversion from input CityGML to Cesium 3D Tile

Table 6: Commands of GeoRocket GeoToolbox

Command	Definition	Input
-i <file>	Input path	./HfT_Innenstadt_LOD2.gml
-o <file>	Output path	./output31463
-s <String>	Strategy how to calculate level of detail - default: kdTree	quadTree
-c <Long>	EPSG code of input	31463

Command	Definition	Input
-m <boolean>	Save metadata	true
-d <boolean>	double sided rendering (no backface culling) - default: false	true

Then, the 3D Tile output is generated in the specified output folder. The specified "quadTree" strategy for the level of detail command will give the outputs in quadtree style from level 0 to 2 which are shown in (Figure 54). This strategy is broadly used for streaming imagery, terrain, vectors, and other data. [95]

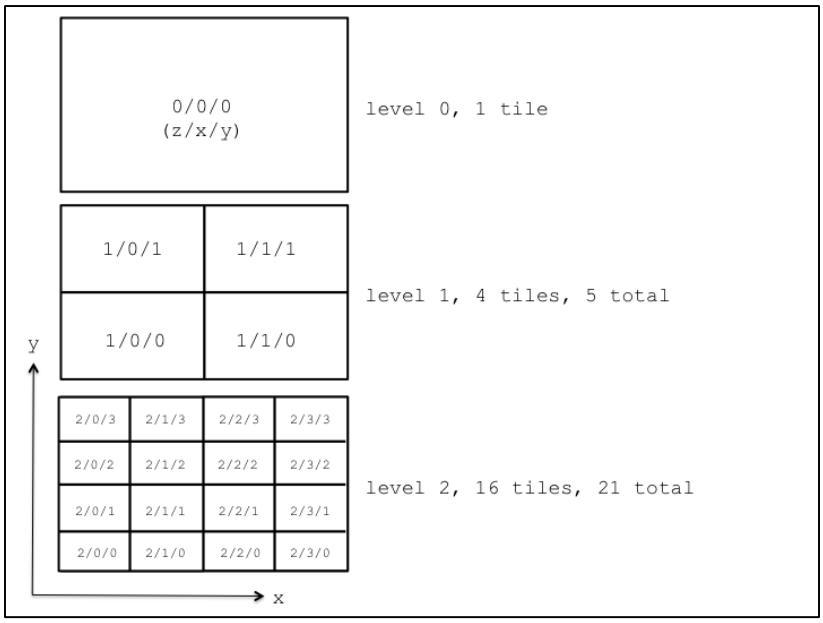


Figure 54: 3D Tile output in Quadtree tyling style (From [95])

4.4.3 Using FME 2017

Another way to convert from CityGML to Cesium 3D Tile is to use the FME Workbench 2017.0. [66] The process is straightforward, the input CityGML is read through the "Add Reader" tool and then the "Cesium 3D Tiles Writer tool" [96] is used by setting the "Add Writer" tool with specified Cesium 3D Tile format as shown in (Figure 55).

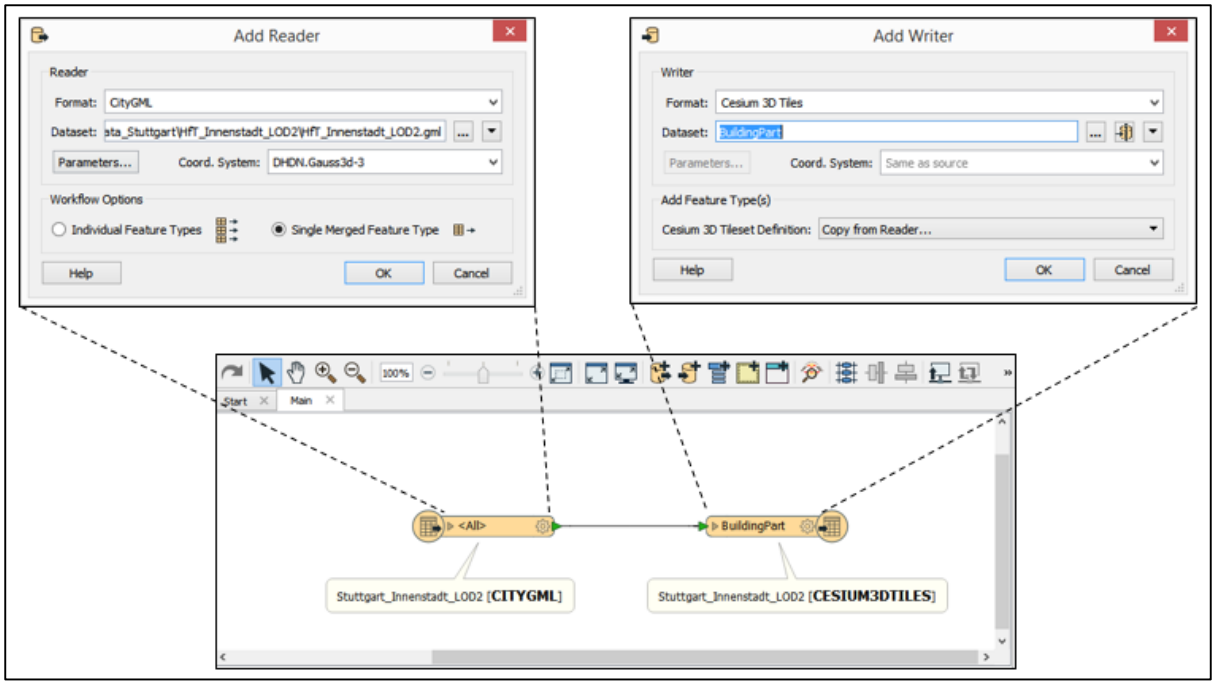


Figure 55: Conversion of CityGML to Cesium 3D Tile using FME 2017

After running the translation, the output file Cesium 3D Tile dataset is in the specified output path and consists of a single JSON file "tileset.json" and a set of b3dm files. The tileset JSON outlines the tile structure of the 3D "b3dm" dataset.

4.5 Developing a 3D Web-based Application using Cesium

The aim of this part is to illustrate on utilizing many sensor data types from the implemented sensor network (Chapter 4.2 and 4.3) and the prepared 3D city model data from (Chapter 4.4). Then, to investigate the performance on the visualization of these data on the client side. The data that are used to build this web application including (1) Data from Ebike-TCU sensors, (2) Data from Garmin Smart Watch, (3) Historical Stuttgart Weather data from OpenWeatherData API and the prepared city models in the area of Stuttgart city.

As stated in (Chapter 3.4 : 3D Web-based Application) the 3D web-based application in this research is built on the 3D globe JavaScript library "CesiumJS" [58] and running on NodeJS server [97]. The Cesium application provides many useful functionalities for this research. Firstly, it is able to show the 3D terrain which is very important for the E-bike application as it will give the reality-experienced of height simulation to the application users. For example, the following figure (Figure 56) shows the comparison of the Cesium application showing the area of Stuttgart city with 2D map with Bing Map Aerial basemap (left) and 3D terrain with attached Bing Map Aerial basemap (right).

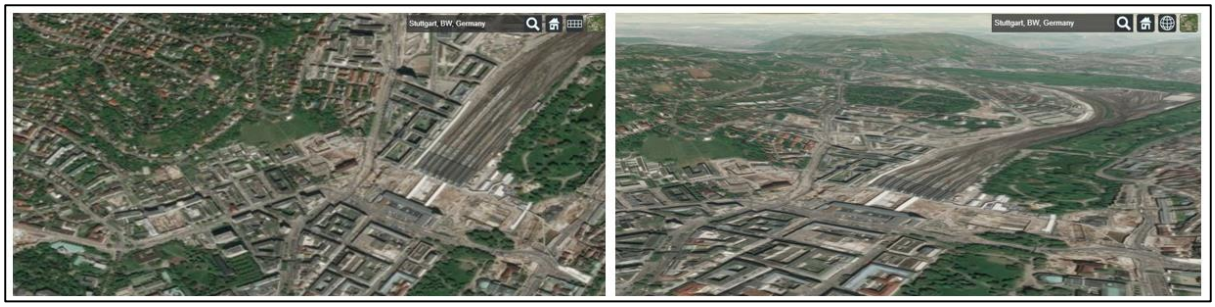


Figure 56: Cesium application showing the area of Stuttgart city with Bing Map Aerial basemap (left) and 3D terrain with attached Bing Map Aerial basemap (right)

Secondly, it contains with many different options of basemap to attach to the terrain as the example shown in (Figure 57) and the users are also allowed to use the basemap from any basemap providers which provided in the supported formats. With this function, it could benefit for the future work of this application by the capability to include the basemap created especially for bike users or any specific purpose. Moreover, it is able to simulate the historical E-bike's route in 3D perspective which the details of this development will be given later on (4.5.4).

Figure 57: Example of the Cesium basic basemap on the 3D perspective in the area of Stuttgart city, (the basemap from left to right: Bing Map Aerial, Bing Map Road, Stamen Toner and Stamen Watercolor.)

The developed web application for this research is named as "i_city ebike sharing". It has the user interface shown in (Figure 58). The basic functions with indicated numbers are described in details below the figure.

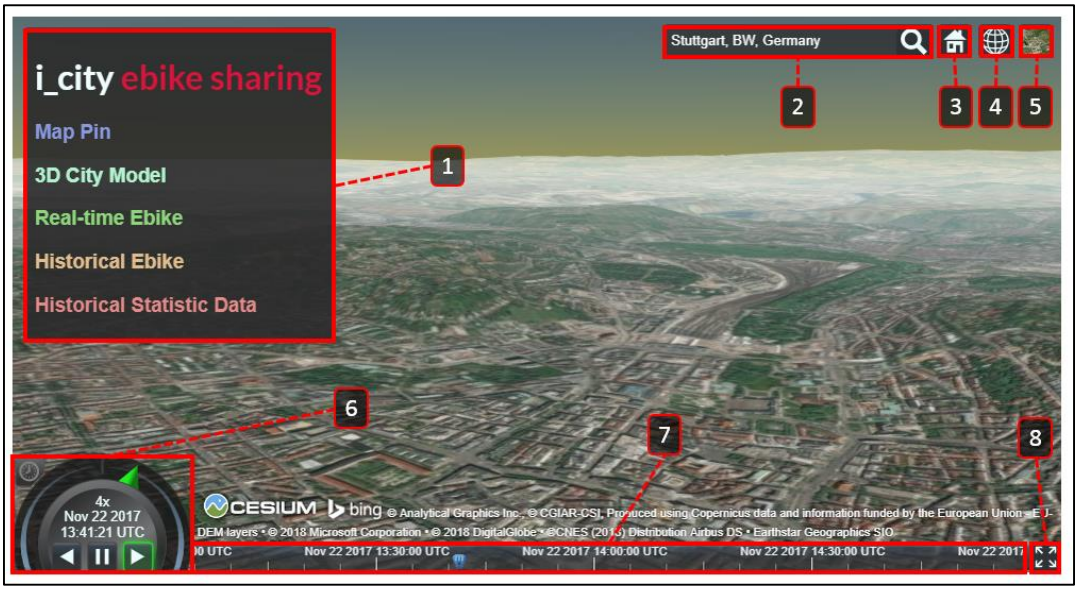


Figure 58: i_city ebike sharing application user interface and its indicated basic functionality number

1. **Main menu**: In this menu, users can select one of the five Sub-Menus to interact some action or display some information window in this applications. The more details are given in (Chapter 4.5.1) to (Chapter 4.5.5).
2. **Geocoder**: It is a location search tool based on the data from Bing Maps. After users indicate the location, the application will zoom to the indicated location.
3. **Home button**: Allow users to zoom back to the default location which is the Stuttgart city.
4. **Scene Mode Picker**: Allow users to switch between 2D and 3D mode.
5. **Base Layer Picker**: Allow users to select basemap.
6. **Animation Clock**: Allow users to control the speed of clock for the animation on the scene.
7. **Time Slider**: Show the current time and allow users to jump to the specific time.
8. **Full-screen Button**: Allow users to switch to the full-screen mode.

To describe the details of each Sub-Menu functionality in the i_city ebike sharing application, the details are given in (Chapter 4.5.1) to (Chapter 4.5.5).

4.5.1 Map Pin

After users click on this Sub-Menu "Map Pin", the small menu will pop-up allow users to show or hide the "Map Pin" which in this application version contains four different categories showing in (Figure 59). The map pin in the first category is "Electric Bike Station – HFT" showing the main E-bike station of the i_city project which currently has only one station. The map pin in the second category is "Bicycle Station" showing the public bicycle station in the Stuttgart city. The map pin in the third

category is "Train Station (S-Bahn/U-Bahn)" showing the train station in the Stuttgart city. The map pin in fourth "Car2Go" showing the Car2GO station in the Stuttgart city.

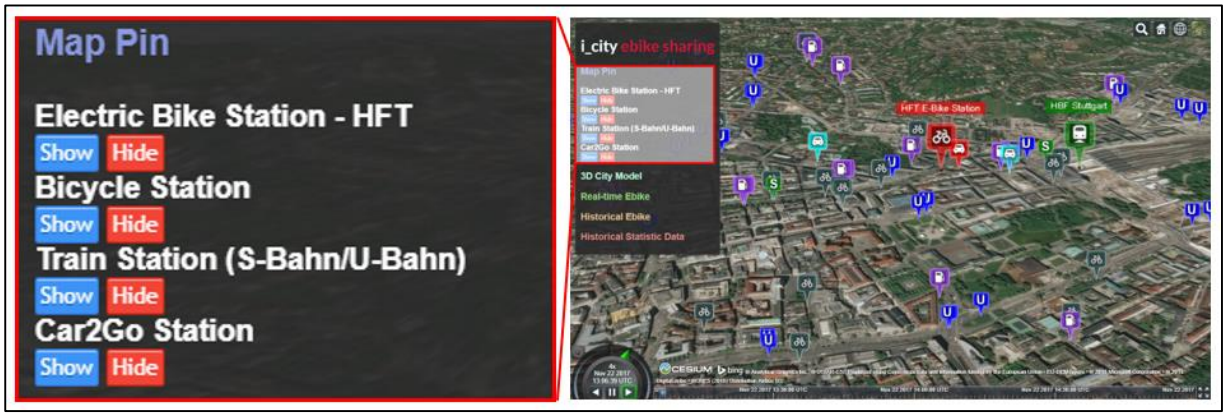


Figure 59: Map Pin Sub-Menu in i_city ebike sharing application

All the Point of Interests (POI) (bicycle stations, train stations, and Car2GO stations) are the data getting from the OpenStreetMap. [98] All the Map-Pin icon symbols are built from MAKI icon set which includes many points of interest symbols and are made for map designers. [99]

4.5.2 3D City Model

After clicking on this Sub-Menu "3D City Model", the small menu will pop-up allow users to show or hide the City Model in the downtown area of Stuttgart city. (Figure 60) The two types of city model are allowed to select including "glTF" and "Cesium 3D Tile" which are the output from Chapter 4.4.

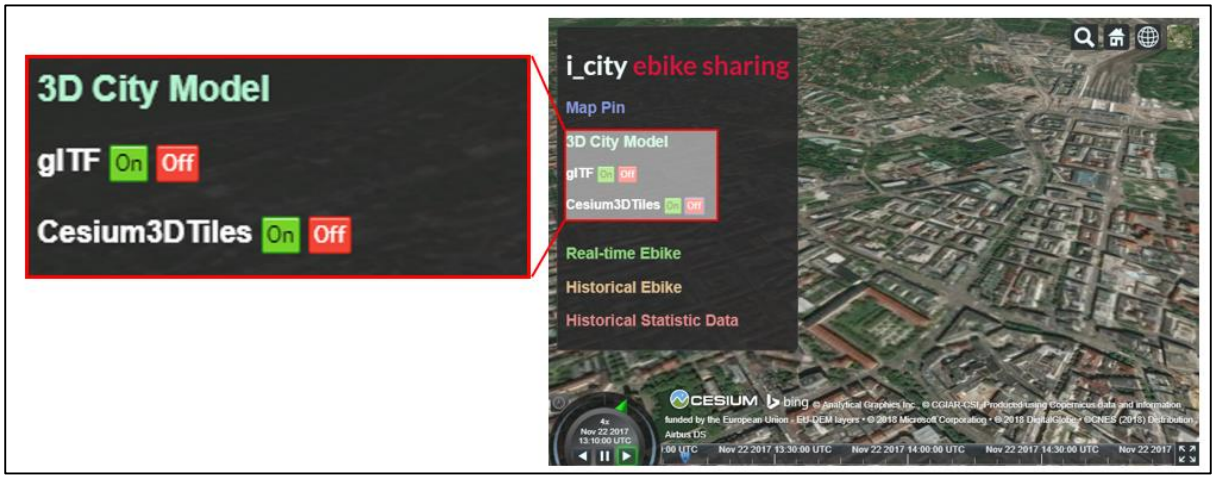


Figure 60: 3D City Model Sub-Menu in i_city ebike sharing application

To show the 3D city model in glTF format, as it is not fully supported by the Cesium library yet, the function "CitydbKmlLayer" from the JavaScript library "3DCityDB-WebClient" [94] is used for

loading the glTF model. The 3D city model in glTF format displaying on Cesium is shown in (Figure 61).

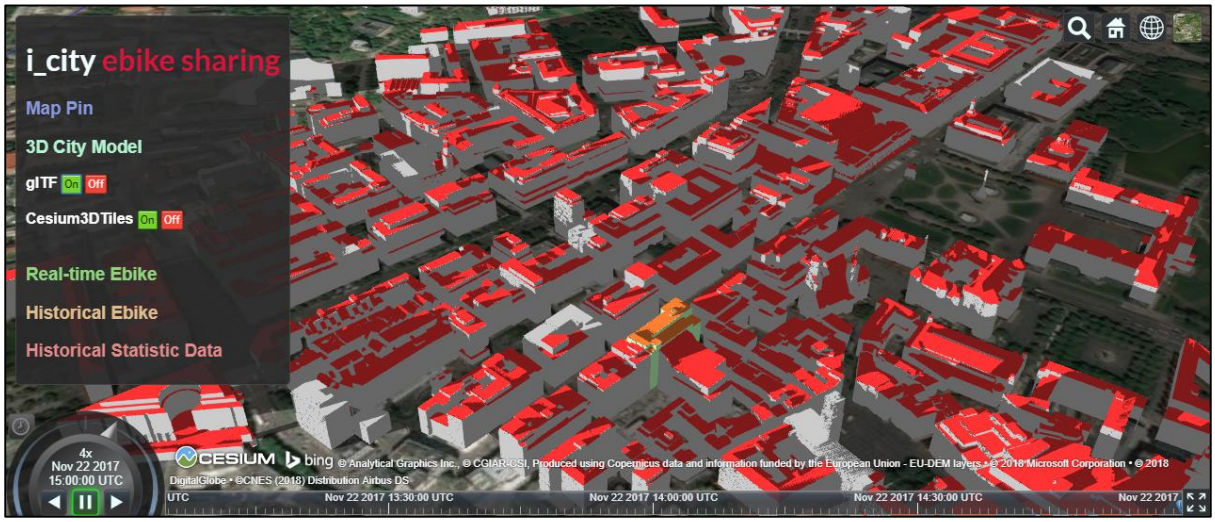


Figure 61: glTF - 3D city model in the downtown area of Stuttgart city in i_city ebike sharing application. (With shadow mode activated at the time of 15:00:00 UTC Nov 22, 2017)

For the 3D city model in Cesium 3D-Tile format, the step to display this dataset is simple as the Cesium library already contain the function to load this city model format to show in the application. Also, while displaying 3D city model in 3D-Tile format, users are able to change the styling of the 3D city model to highlight the other features of the application. The following figures show the example of 3D city model of Stuttgart city with the default styling and the black-transparent styling respectively.

Figure 62: 3D city model in Cesium 3DTile format with default styling in the downtown area of Stuttgart city in i_city ebike sharing application.

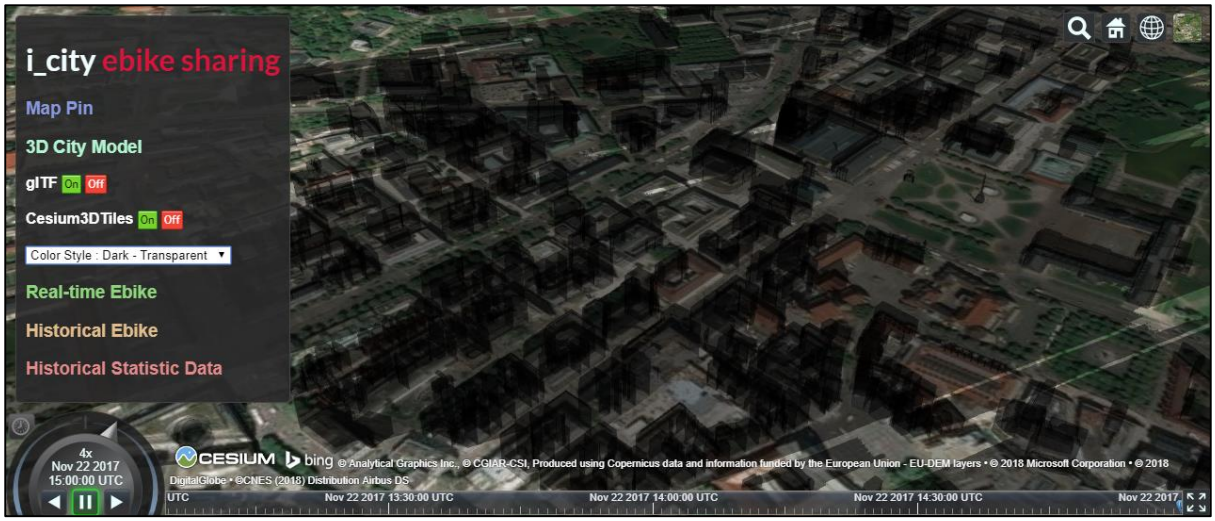


Figure 63: 3D city model in Cesium 3DTile format with black-transparent styling in the downtown area of Stuttgart city in i_city ebike sharing application.

4.5.3 *Real-time E-bike*

With this Sub-Menu "Real-time Ebike", users are allowed to request the latest information of each E-bike. The development of this function uses the JQuery library [100] to request the JSON data from the sensor network. As the final sensor network in this research is SensorThings API, the request for the latest updated value on each "Datastream" entity can be done simply with a single request. For example, in the application script, the following JavaScript is used to request the result (Observation) from the "Datastream" entity ID 1. The query operators using in the request are including "$orderby = resultTime desc" to order the response result body from the latest result time and "$top = 1" to get only the first latest available result value. (Figure 64) The full script of the program to make all requests by user choice is attached on the Appendix disc. (Appendix C)

```
$.getJSON(
"http://localhost:8080/SensorThingsService/v1.0/Datastreams(1)/Observations?$orderby=result
Time%20desc&$top=1", function( data ) {
        // do something with "data"
        // data => Latest value of the observation result of Datastream id 1
});
```

Figure 64: Example of the JavaScirpt request for the latest result of the Datastream entity ID 1 (E-bike01 fuel level)

The details on using this functionality are described in (Figure 65) and the instructions below the figure.

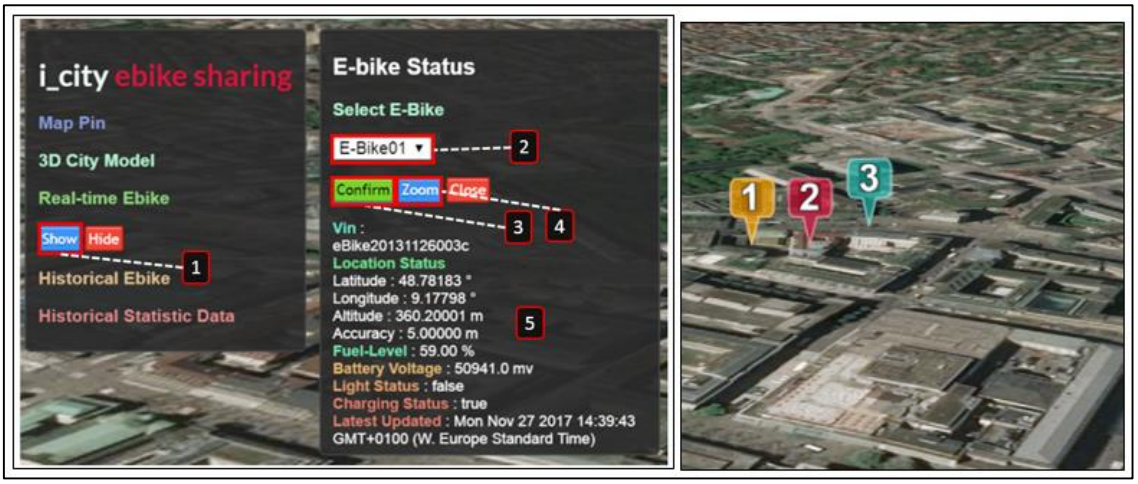


Figure 65: Real-time Ebike Sub-Menu instruction

Real-time Ebike Sub-Menu Instruction

(The instruction bullet numbers match with the number indicated in Figure 65.)

1. Click on "Show" button and then the E-bike Status will pop-up on the right side as shown in (Figure 65 – left).
2. Select the "E-bike" number, in this research, there are currently four available E-bikes which the list of these E-bikes are shown in (Appendix B - 5).
3. Click on "Confirm" button to confirm the selection E-bike.
4. Click on "Zoom" button to fly from the current view to the location of selected E-bike as shown in (Figure 65 – right).
5. After click on "Confirm" button (Step 3), the latest details of the selected E-bike are shown in this area.

4.5.4 Historical E-bike

With this Sub-Menu "Historical Ebike", users are allowed to request the historical E-bike session simulation on the selected session date. For the usage of this functionality, the details are in the figure with the instructions below. (Figure 66)

Figure 66: Historical Ebike Sub-Menu instruction

Historical Ebike Sub-Menu Instruction

(The instruction bullet numbers match with the number indicated in Figure 66.)

1. From this drop-down list menu, users can select the E-bike session.
2. Here, users can click on the "confirm" button to load the E-bike session simulation on the screen. On the background, the E-Bike CZML is generated and displayed on the screen on the client side. After loading is complete, the application clock on the lower left corner and the time-slider range on the bottom of the screen will change to match the time of the E-bike session. The default clock speed in this application is 20 times faster than normal clock speed.

   Or, users can click on the "remove" button to remove the E-Bike simulation on the screen.
3. In this section, users can select on "E-Bike track" option to change the camera to track the E-Bike simulation or select on "Free" option to turn back the camera to normal mode at the starting position.
4. The E-bike simulation is shown on the application. The position of the E-bike is shown as a sphere in red color together with the indicated label floating on the top-right of the E-bike position. The path of the E-bike that is already visited is shown as a green polyline on the terrain.
5. The E-bike usage information in this window shows the current status matching the E-bike status on the specific time and the application clock (shown in the lower left corner of the application).

The development of this functionality is made by building a CZML from the specified E-Bike usage data from the server. The CZML (Cesium Language) is a JSON describing a time-dynamic primarily for display in a Cesium application. [56] The JQuery library [100] is used to request the specified observation result to build up the CZML on the client side later on. With the SensorThings API standard, users can request the observation result with the specified period of time by using the filter query ($filter) and also limit the response body to return only "result" and "resultTime" key-pair-values by using the selected query ($select). For example, a figure below shows the request for the observation result of Datastream ID 1 (Fuel level of E-bike01) from 13:00 to 15:00 on 22nd November 2017. (Figure 67)

```
$.getJSON( "http://localhost:8080/SensorThingsService/v1.0/Datastreams(1)/Observations
$orderby=resultTime&
$filter=resultTime ge 2017-11-22T13:00:00.000Z and
        resultTime le 2017-11-22T15:00:00.000Z &
$select=result,resultTime" , function( data ) {
      //do something with the result data
});
```

Figure 67: Example of the JavaScirpt request for the result of the Datastream entity ID 1 (E-bike01 fuel level) from 13:00 to 15:00 on 22nd November 2017

The challenging part on this functionality development is to show the "E-bike usage information" (Figure 66 – Right window) related to the application clock which users can freely adjust this value on the client side. In order to implement this step, the application must be able to fill the gap of data by making an interpolation over-time. To minimize the computation power on the client side, the interval of 500 milliseconds is used to compute each interpolated value.

4.5.5 Historical Statistic Data

With this Sub-Menu "Historical Statistic Data", users are allowed to request the statistic of the historical E-bike usage with a specific time range. For the usage of this functionality, the details are in the figure with the instructions below. (Figure 68)

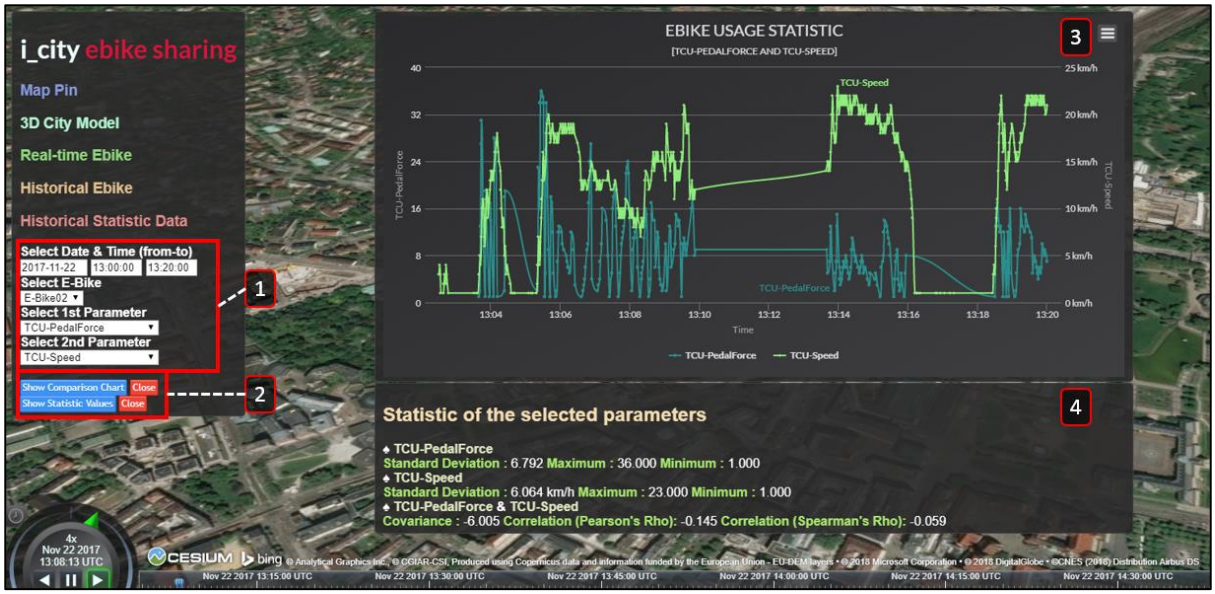


Figure 68: Historical Statistic Data Sub-Menu instruction

Historical Statistic Data Sub-Menu instruction

(The instruction bullet numbers match with the number indicated on Figure 68.)

1. From this panel, users can select date & time for the beginning and the end of for the request usage time. On the first selector, users can type for the specific data in [yyyy-dd-mm] format or click to show the available date to select. On the second and third selectors, the user can type for the specific time in [hh:mm:ss] format or click to show the dropdown list of the time in the interval of 10 minutes. The example of the first, second and third selectors are shown in (Figure 69 – left, middle-left, middle-right) Then, users can select on the interested E-bike ID on the next selector. Next, users can select the two interested parameters to show up in the statistic chart and information windows. For example, the available parameters are shown in (Figure 69 – right).

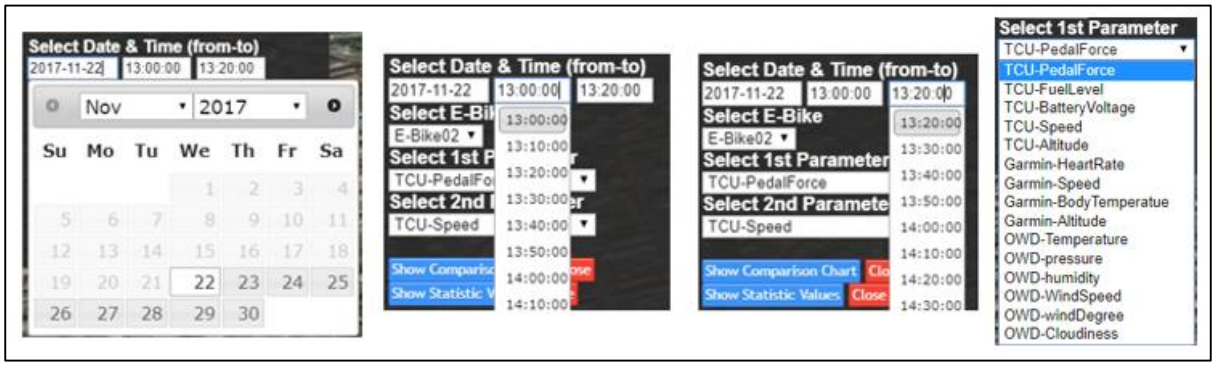


Figure 69: Data and Time selector for historical statistic data of the E-bike usage

2. In this section, users can click on "Show Comparison Chart" button to open the chart window (section 3) or click on "Show Statistic Window" button to open the statistic window (section 4). Users can click on "Close" button anytime to close these pop-up windows.
3. This pop-up window shows the E-bike usage statistic chart of the two selected parameters from section 1. Users are able to interact with this chart like zoom in to the selected time period by click and drag on the chart, to see the specific value of each parameter by hover the mouse on top of the chart (Figure 70 – 3.1), and to click on the top right of the chart to print out the chart or to save the chart with preferred format (Figure 70 – 3.2).

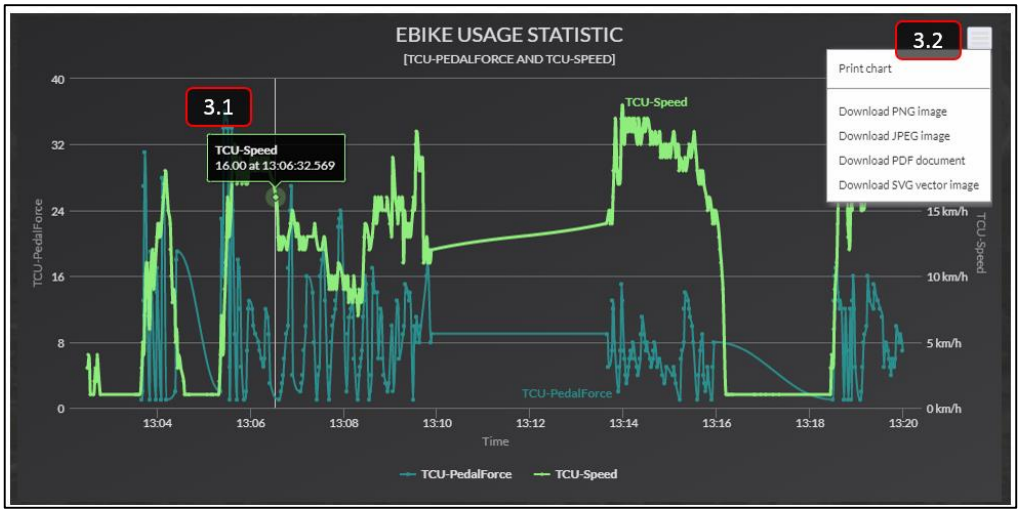


Figure 70: E-bike usage statistic chart

4. This pop-up window shows the E-bike usage statistic values of the two selected parameters from section 1. The statistic values are including the values of "Standard Deviation", "Maximum", and "Minimum" of each parameter and also the correlation statistic of these two selected parameters which are including "Covariance", "Pearson's Correlation Coefficient" and "Spearman's Correlation Coefficient"

The development of this functionality is built-up based on the sensor data from the implemented sensor network which the algorithm to request for the data is same as in (Chapter 4.5.4). The specified date, time, E-bike id, and two parameter values from the selectors are used to make a request for the sensor data. These date&time selectors are built from jQuery Timepicker and Datepicker library. [101] While the other selectors are built with simple HTML dropdown list. For the chart window (Figure 68 – section 3), it is built up with the Highcharts JavaScript library. [46] And for the statistics window (Figure 68 – section 4), the statistical data are computed with the "JStat" - the JavaScript Statistical Library. [102]

5. Evaluation

In this research, three different sensor network protocols have been implemented and evaluated to select the most qualified protocol for monitoring the E-bike usage in a 3D application. The sensor data using in this research are including 1) the data from the SMART E-bike, 2) the smartwatch Garmin Fenix X5, and 3) the temperature data from OpenWeatherMap. The number of sensor datastreams imported to the sensor network is shown in (Table 7).

Table 7: Number of Datastreams using in this research

Date	Total Datastreams
22 November 2017	97,981
24 November 2017	167,193
27 November 2017	45,818
4 December 2017	89,313

To evaluate the technical performance on each sensor network protocol, the following metrics are used including 1) Request size of an operation, 2) Response time of an operation, 3) Response length of an operation, and 4) Support of dynamic 3D location of a moving sensor. The overall results of the evaluation are shown in (Table 8). The details on each evaluation metric will be described from Chapter 5.1 to 5.4.

Table 8: Technical performance evaluation results

Evaluation Metrics	OGC Sensor Observation Service	OGC SensorThings API	Advanced Sensor Data Delivery Service
Request size of an operation	✓	✓✓	✓✓
Response Time of an Operation	✓	✓✓	✓✓
Response Length of an Operation	✓✓	✓	✓
Support of the Dynamic 3D Location of a Moving Sensor	✓	✓✓	x

(✓✓ = Good support, ✓ = support, x = not support directly and more development is needed)

5.1 Request Size of an Operation

In IoT development, the payload packet size is very important and needed to be considered. [8, 103] To analyze the capability of all sensor network protocols, the request body for the observation results from different sensor server are compared. The requests are written with the built-in browser

jQuery library and the example of the request are shown in the following figures. (Figure 71, Figure 72, Figure 73). [100]

```
$.getJSON("http://localhost:8080/SensorThingsService/v1.0/Datastreams(<Datastreams id
here>)/Observations", function (data) {
    // do something with the response "data"
});
```

Figure 71: Request function to get the observation results from SensorThings API server

```
$.post("http://localhost:8080/52n-sos-webapp/service", {
    request: "GetResult",
    service: "SOS",
    version: "2.0.0",
    offering: "http://localhost:8080/52n-sos-webapp/offering<offering id here>",
    observedProperty: "http://localhost:8080/52n-sos-
webapp/observedProperty<observedProperty id here>"
}, function (data) {
   // do something with the response "data"
}, "json");
```

Figure 72: Request function to get the observation results from Sensor Observation Service server

```
$.getJSON("http://localhost:8080/api/observation?sensorName=Test", function (data) {
    // do something with the response "data"
});
```

Figure 73: Request function to get the observation results from ASDDS

As a result, it shows that the SOS protocol needs a larger size of code and transaction on each request which is matched with the previous research on comparing the different network system [8], which shows that the requests of the SOS protocol are at least 47% larger than other protocols. Moreover, it shows that with the jQuery library, there is no need to specify any header and body to request the particular observation JSON result array from the SensorThings API and ASDDS server.

5.2 Response Time of an Operation

Server response time is the time taken for the server to respond to the client's request. It is an important factor in the final web application speed and performance. [104] In this research, the response time of a single request has been evaluated and compared. A simple Node-JS application on the server-side is written to perform a request on one observation result and print the response time to the console window. The algorithm to check the request time are shown in (Figure 74). The operations are done 10 times on the same CPU and network capacity. All results are shown in (Table 9).

```
//==============Input paramerters here!!==========================
var url = "";            // input test URL here
var headers = {};        //input test header here
var method = ""          // input test method here
var body = ""            // input body here (Only for SOS)
//================================================================
var request = require('request');
function RequestTimeChecker() {
    let options = {
        url: url,
        headers: headers,
        method: method,
        time: true,
        body: body
    }
    request(options, function (error, httpResponse, body) {
        console.log('Request time in ms :', httpResponse.elapsedTime);
        console.log('Body :', body);
    })
}
RequestTimeChecker();
```

Figure 74: Node-JS application on the server-side to check the response time on each server network.

Table 9: Response time for a single observation from different sensor server.

Attempt	Response time for a single request (millisecond)		
	SensorThings API	SOS	ASDDS
1	19	40	20
2	18	35	21
3	19	41	20
4	19	38	20
5	18	45	20
6	18	34	21
7	19	49	20
8	18	45	21
9	18	41	20
10	18	34	20
Avg	**18.4**	**40.2**	**20.3**
Max	**19**	**49**	**21**
Min	**18**	**34**	**20**

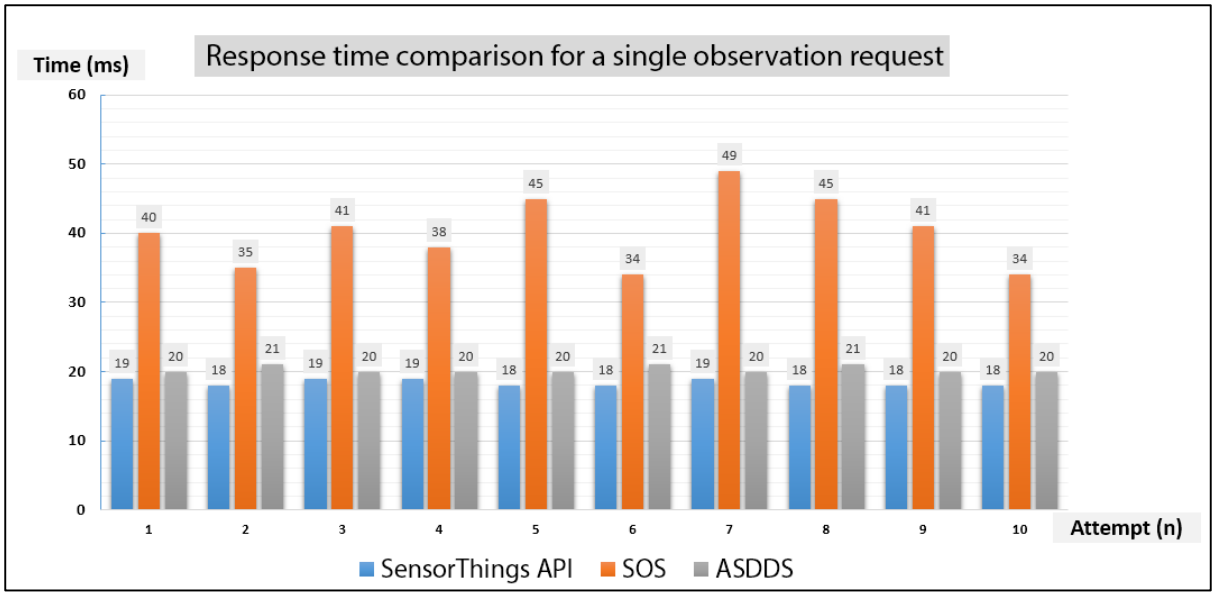


Figure 75: Response time comparison for a single observation request

From the response time result, it is clearly seen that the SOS protocol takes the longest time until the response comes back. Although the highest different of the response time between SOS and SensorThings API protocol from 10 attempts is only 30 milliseconds (from the 7th attempt, Figure 75 and Table 9 - SOS used 49 milliseconds and SensorThings API used 19 milliseconds), but later on in the 3D visual analytics application on the client side, the number requests and responses will be at a high rate which will significantly reduce the application performance.

5.3 Response Length of an Operation

The response length is also an important part as it will consider the amount of data transferred from the server side to the client side. In some previous work, the response length of the SOS provides much longer response length as an XML format if comparing to the other JSON-based standard for example in [8]. But, with the 52North SOS 4 implementation version, the response set of the value and time are available in JSON format and are embedded in the single response file. For example, the (Figure 76) shows the response body from the SOS server. While the OGC SensorThings API and ASDDS provide the response as a repetitive attribute key-value pairs as shown in (Figure 77).

```
{
  "request" : "GetResult",
  "version" : "2.0.0",
  "service" : "SOS",
  "resultValues" : "
                15#2012-11-19T12:30:00.000Z,
                159.15#2012-11-19T12:31:00.000Z,
                159.15#2012-11-19T12:32:00.000Z
                "
}
```

Figure 76: the response body from the 52North SOS server

```
{
"value": [
    {
      "resultTime": "2017-11-22T17:20:15.041Z",
      "result": 67
    },
    {
      "resultTime": "2017-11-22T17:20:05.032Z",
      "result": 63
    },
    {
      "resultTime": "2017-11-22T17:19:54.778Z",
      "result": 60
    }
    ]
}
```

Figure 77: JSON response with repetitive attribute-value pairs from ASDDS or SensorThings API

However, to compensate this repetitive drawback in SensorThings API, firstly, the users are capable to request only the needed attributed value, for example, only "result" value with the query capability over the http request. Secondly, the SensorThings API is supported by service-driven "pagination" which the number of response body can be limited on each request. If the response size is larger than the limited number on the server-side, then the "nextLink" annotation is included in a response which represents the link to the next part of the whole response body. With this pagination functionality, the developer can freely control the response JSON body size on each request. Also, in the developer perspective, it is good for them that the response body provide the key-value pairs as it helps to understand the response structure and reduce the chance to incorrectly use the data.

5.4 Support of the Dynamic 3D Location of a Moving Sensor

In this research, the feature to support the moving sensor is extremely important as the location of sensors for E-bike are always dynamically change over the usage period. Accordingly, the sensor protocol needs to support the 3D-locations of the sensors directly without any conflict.

For the OGC SensorThings, the 3D locations are supported in GeoJSON format. The location of the "Things" entity can be updated through the HTTP PATCH method. The following figure (Figure 78) shows the example URL of the "Thing's location" in SensorThings API standard and the example

of updated request body is shown in (Figure 79). After the geospatial location of the "Thing" entities has been updated, the existing location of the "Locations" entities will be collected in the "HistoricalLocations" entity.

```
http://(hostname):(port)/(STA name)/v1.0/Things(id)/Locations
```

Figure 78: SensorThings API URL to access the Thing's location

```
{
"location": {
      "type": "Point",
      "coordinates":
      [9.1726,
        48.7803,
        250]
    }
}
```

Figure 79 : Example of the updated request body for the Things' Location of SensorThings API

For the OGC SOS, the observation with the geospatial location is supported by specifying the sampling geometry of the observation in the GML based. The example of the observation of the E-bike fuel level with the 3D location is shown in (Figure 80) which is the requested in XML based. To compare with the SensorThings API request body for updating the geospatial location (Figure 79), it is clearly that the request body of the SOS geospatial location is much longer to build up as in the SOS, it is always needed to clarify the GML specification, for example, "srsName" and "srsDimension" in XML based.

```
<om:OM_Observation gml:id="obsTest1"
    xmlns:om="http://www.opengis.net/om/2.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:gml="http://www.opengis.net/gml/3.2"
   xsi:schemaLocation="http://www.opengis.net/om/2.0
   http://schemas.opengis.net/om/2.0/observation.xsd">
    <gml:description>Spatial observation test instance: Ebike Fuel Level</gml:description>
    <gml:name>Spatial observation test 1</gml:name>
    <om:phenomenonTime>
        <gml:TimeInstant gml:id="pt1">
            <gml:timePosition>2017-11-22T13:00:00.00</gml:timePosition>
        </gml:TimeInstant>
    </om:phenomenonTime>
    <om:resultTime xlink:href="#pt1"/>
    <om:procedure
   xlink:href="http://mySOSURL?service=SOS&amp;request=DescribeSensor&amp;version=2.0.0&am
   p;procedureIdentifier=&quot;procedure1"/>
    <om:parameter>
        <om:NamedValue>
            <om:name xlink:href="http://www.opengis.net/def/param-name/OGC-
   OM/2.0/samplingGeometry"/>
            <om:value>
                <gml:Point gml:id="SamplingPoint">
                    <gml:pos srsDimension="3"
   srsName="urn:ogc:def:crs:EPSG:7.9:31463">3512583.00 5404142.00 0.00</gml:pos>
                </gml:Point>
            </om:value>
        </om:NamedValue>
    </om:parameter>
    <om:observedProperty
   xlink:href="http://sweet.jpl.nasa.gov/2.0/hydroSurface.owl#WaterHeight"/>
    <om:featureOfInterest
   xlink:href="http://wfs.example.org?request=getFeature&amp;featureid=river1 "/>
    <om:result xsi:type="gml:MeasureType" uom="cm">28</om:result>
</om:OM_Observation>
```

Figure 80: Example of SOS Observation with spatial location indicated in a parameter

For the ASDDS, there are no features or entities to support the 2D or 3D locations of the sensor systems directly. The possible way of the current version of the ASDDS is to add the location entities (for example: Latitude, Longitude, Height) as the new observation entities. Consequently, the more requests in terms of number and length are needed as one updated geospatial location in 3D will need at least 3 times requests for each dimension of the location values.

6. Results

6.1 Discussion

6.1.1 Sensor Network

During the last six months, the three sensor network protocols: 1) OGC SensorThings API, 2) OGC Sensor Observation Service (SOS), and 3) Advanced Sensor Data Delivery Service (ASDDS) have been implemented and compared to find the most qualified protocol for managing the heterogeneous sensor systems in the i_city E-bike Sharing Project.

On the implementation difficulty, the OGC SensorThings API, one of the OGC standard from Feb 2016 [5], has many sources of descriptions, practical application examples, and tutorials for new IoT developers in clear and understandable format. So that, it is the simplest framework to implement among all three candidate protocol. On the other hand, the OGC Sensor Observation Service has one critical disadvantage. Although it is based on the OGC's Sensor Web Enablement, which has been continually developing by a big group of researching for more than decades, it is the most difficult standards to learn. The complexity of the structure of the request body requirement in XML-base hinders the implementation of the project. On the other hand, the ASDDS structure is less complicated and can be implemented less difficulty. The fact that it is developed by HFT Stuttgart makes it easier to get some guidelines by personal contact. [34] Nevertheless, all available documents are in German language and the number of available application examples is very limited.

For the database structure, both OGC SensorThings API and OGC Sensor Observation Service protocols are developed based on the OGC and ISO Observations & Measurements 2.0 standard. Their relational connections in the database structure can model and refer to the monitoring systems in the real world with a complete metadata. For, ASDDS, in contrast, some entities are missing, for example, the support entities to collect the geospatial location and the metadata of both sensors and observations. Moreover, the one-to-one entities relationship in the ASDDS impacts some of the repetitions of data on the server-side.

To apply the sensor protocol in the i_city E-bike sharing project, there was an unexpected issue of the E-bike's TCU sensor limitation that it will send the data through the server with the limited message length so that when the device sends a message to the server, some data attributes will not be sent in the same message but keep approximately 1-1.5 second delay and be sent as a separate message in order to keep the least message length communication. Accordingly, data will be sent to the server only the latest change value of the TCU sensor. For my finding, the OGC SensorThings API is the best IoT sensor protocol candidate to deal with this non-pattern irregular time-series data as the geospatial

data in SensorThings API standard are collected in the "Locations" and "HistoricalLocation" entities which are independent of other observation results in "Observation" entities, so that the result of the updated geospatial location can be updated independently. On the other hands, the OGC SOS is collecting the data from moving sensor with the "OM_SpatialObservation" according to the "Spatial Filtering Profile" of the O&M standard. It always collects the observation result with the update of the location so that the locations will be sent only when there is an update of some other observation result. In conclusion, the SensorThings API can deal with this issue in a smooth way, while the ASDDS has no support of the geospatial locations but they must be added to the server as a new observation type in the network.

Throughout the application development phase, the OGC SensorThings API has proven to be a clear and efficient solution to receive the sensor data in the very compact request form. With a single HTTP request, users can flexibly request the very detailed response body of each observation with a query capability. For example, "$expand", "select", "$stop", "$filter", etc. For "$expand", it is used for requesting the result time, phenomenon time, latest and historical geospatial location, and all its meta data about things, sensors, and features of interest. For "$select", it is used for limiting the response body so that only some part will be shown. If users want to narrow the response by limiting only the highest or the lowest values over a specific period of time, they can use "$top" and "$filter". On the other hand, the OGC Sensor Observation Service always needs the large size of the request body to the server to request any message. For the ASDDS, the response is simple and similar to the OGC SensorThings API, however, it still lacks the capabilities to request for complex queries.

Considering the aggregation function over the sensor network, only the ASDDS protocol has this feature as its built-in function, but in this ASDDS version, it still supports only the "average" function with the limit option for the time-range selection. For the OGC SensorThings API and the OGC SOS, while they originally do not have this function, there are several methods to add it. The first way is to add the aggregation function output as a pre-computed value on the server-side. The second way is to add some programming library to compute this value on the client-side. The last way is to add an aggregation function on the server-side.

As the extensibility of the sensors is being examined, one disadvantage of the OGC SOS is found. This protocol needs a specific interface with a large request body for each sensor to add a new sensor or an IoT device to the network. In contrast, the OGC SensorThings API has proven the simple way to add a new sensor to the network with any simple HTTP request. Moreover, the OGC SensorThings can define physical any objects and any IoT sensors separately by specifying a "Thing" and a "Sensor". In other words, one physical object "Thing" can have one or many "Sensor(s)" and this protocol is able to detect each of them. For example, in future development, one E-bike can have not

only TCUs, but also GNSS sensor, IMU, and any other IoT devices to observe many properties of the same E-bike at the same time.

6.1.2 3D Web-based Application

A 3D web-based application has been developed in the final part of this research. The Cesium JavaScript is used to build the main part of the application. For the 3D city model, the current cesium application version is not supported to show the city model in CityGML file format directly. Some additional steps for file conversion to Cesium 3D-Tile or glTF are needed. Accordingly, the conversion of CityGML to Cesium 3D-Tile and glTF is done in this work using different tools.

At this stage, three different tools are deliberately examined to find out the best solution. By using the open-source 3DCityDB (Chapter 4.4.1), the tool has proven capabilities to import the CityGML data in PostgreSQL database and export to the glTF file format with a defined styling option such as roof and wall color of the building. A drawback of this tool is that it still needs some manual steps for users to import the data into the database first before exporting the data to a preferred format. There is no single step to converse the CityGML data to glTF file format directly. On the other hand, the GeoRocket GeoToolbox 1.0.2 (Chapter 4.4.2), which is a lite source program written in Java for converting the input CityGML file into Cesium 3D-Tile, can run the conversion through the command window in a single step without exchanging file in the database. The Cesium 3D-Tile output from the Stuttgart CityGML dataset can be shown on the Cesium 3D application without any problem. Similarly, the FME 2017 (Chapter 4.4.3) can convert a CityGML into Cesium 3D-Tile with one step and show the result on the Cesium application as well. In addition, this tool can view and edit the model of CityGML in both 2D and 3D perspectives with a simple user-interface as shown in the example in (Chapter 3.5). However, the desktop program of this tool must be installed first.

For the web-based visualization of the 3D city model in the Cesium application, the result in Cesium 3D-Tile format has proven that it takes less time and resource on the client side to load compared to the file in glTF format. Still, at the moment, there is no open source tool to convert data from CityGML to Cesium 3D-Tile directly. Currently, Cesium 3D-Tile is under the stage of consideration where it will be certified as one of the OGC community standards under the 3D Portrayal Service. This will allow Cesium 3D-Tile to become interoperable with other 3D geospatial formats such as X3D, glTF, I3S, and CityGML. [105, 106]

For the application development process using Cesium JavaScript library, it is simple to learn how to use the Cesium framework as there are many examples and tutorials on the website [58] and also a big developer community. To simulate the historical E-bike route, the CZML file format is built

on the client-side by requesting the sensor data from the sensor network to show 3D E-bike path which shows the good results on the application (Chapter 4.5.4).

6.2 Conclusion

This thesis research proposes the evaluation of the implementation of three different sensor protocols including 1) OGC SensorThings API, 2) OGC SOS, and 3) Advanced Sensor Data Delivery Service (ASDDS) to integrate the heterogeneous sensor systems together and the evaluation of data sufficiency provided by each protocol. To summarize the finding of this research, the OGC SensorThings API is the most competent IoT framework among the three candidate protocols. During the implementation process, the OGC SensorThings API has great quantities of documentation and tutorials. In fact, those samples are also the most detailed compared to the others. [36]. Also, the communication of OGC SensorThings API among IoT devices, servers, and clients is in a standards-based REST style and JSON-based encoding providing its advantage for developers to understand, implement, and develop an application based on this standard in a compact way. Additionally, it is the only protocol among the three candidate protocol that supports the MQTT communication which is very useful for adding a light-weight IoT device in the future. Even though the lack of the aggregation function is its downside, there are several methods to add this feature to this protocol including 1) adding an aggregation function on a client-side, 2) adding an aggregation function a server-side, and 3) pre-computation of the aggregated results on the server-side.

In conclusion, the OGC SensorThings API sensor framework serves all aspects including the IoT interoperability, understandability, and maintainability for developers and researchers which lead to the future expansion and development in terms of IoT devices and applications. It is also a successful sensor framework in the implementation for visualization and analysis of the E-bike usage in 3D City model by integration of heterogeneous sensor data.

6.3 Future Work

6.3.1 Sensor Network

The finding of this research concludes that the most qualified sensor protocol to manage the sensor data in this project is the OGC SensorThings API framework. The client-side application can make a request for the observation data in JSON format with a simple HTTP request. Consequently, it supports a set of built-in filter operations for comparison and logical query. However, it still lacks of the aggregation operator such as an average function on the server side. So that, this research uses a JavaScript library to compute such a function which need to load all data in defined time range on the

client side first. Future work in this part could be a real-time connection from both E-bike sensor and Garmin Smart Watch to the i_city sensor network.

Furthermore, there is a limitation on a connection of the sensor devices to the sensor network in real-time. For the E-bike data, there is a delay in a deployment of the E-bike sensor network due to the security issue. Currently, the E-bike data is uploaded to the server every 24-hour period with the FTP solution. For the Garmin Smart Watch, the data are sent through the Garmin service called "Garmin Connect" where the raw data in GPX and TCX formats have to be extracted from this service manually. Future work in this part could be a real-time connection from all E-bike sensors and Garmin Smart Watches to the i_city sensor network.

Another point that can be worked on is about data collecting rules. In this project, all the sensor data are collected into the sensor network without any constraint. For further research, some rules or constraints can be applied to validate the accuracy of its source, for example, the location feasibility if the movement speed of the sensor is realistic for E-bike. Hence, any observation data of the sensor position from one location to another with excessed defined speed will be rejected by the server. In addition, some observed property data are provided by both the E-bike sensor and the Garmin Smart Watch in parallel. Nevertheless, these duplicated data are not being deployed to improve the accuracy or to observe which data source provides more precise data. The future work of this part could be to find a method to improve the data quality by data analysis and adjustment.

Lastly, the sensor system in this project is still open for more IoT devices in daily-life such as smartphones, wearable health sensor devices, etc. These devices could measure some missing important parameter that the Garmin Smart Watches and the E-bike sensors do not collect during the usage time. Furthermore, any supports of a sensor network for various IoT products such as Apple iPhone, Samsung Galaxy, Apple Watch, etc., would give a good impression for users after the E-bike sharing project is deployed in a city level. From a researcher point of view, more sensors equipped on each E-bikes means more useful information. In other words, greater opportunity for researchers and developers to analyze and utilize those data in a Smart Cities concept. For instance, an IMU sensor equipped on E-bike would give a parameter to calculate the real terrain slope over the whole usage time; or a temperature sensor to measure the outside temperature and then the relation of this value and the body temperature from the wearable device can be computed. Moreover, some type of sensors or devices can be integrated to let the users identify their simple information like age and gender which are important and useful for research purpose.

6.3.2 3D Web-based Application

The final 3D web-based application in this research is built using the Cesium JavaScript library. There are two possible methods to simulate the E-bike usage route in the application. The first method which is also using in this research is to request the data from the sensor server and built-up the data in CZML format on the client-side. Another method is to prepare the data in CZML format in the server-side and let user load this dataset directly. The future study could be conducted to compare the performance of both methods to simulate many E-bike routes at a time. In the part of a 3D simulation of the E-bike path, this research uses the data from the E-bike data collection in the i_city project which currently has only one participant taking a ride at a time. Consequently, the E-bike simulation on the application has only one path at a time. In the next step where there are more participants using E-bikes at the same time, the 3D simulation should be developed to support multi-paths display while keeping its performance on the client side.

In a context of 3D city model, the application is able to show the city models on the web. However, so far it has been utilized only as a visualization purpose. The future work for this part could be using the 3D city model to analyze some valuable information to users. For example, the "Sunny Side" project [107] shows developed application based on the spatial analysis of 3D city model which provides users the time-range of the sunny area at the specific location. This project's idea can be combined with the E-bike sharing project to create a function to predict sunlight and shade against the E-bike path at a time based on 3D city model.

References

[1] Hochschule für Technik Stuttgart, *Projekte Intelligent city: Forschung an Fachhochschulen, Starke Fachhochschulen - Impuls für die Region.* [Online] Available: https://www.hft-stuttgart.de/Forschung/Kompetenzen/ZATB/Projekte/Projekt195.html/en. Accessed on: Dec. 08 2017.

[2] Hochschule für Technik Stuttgart, *i_city: intelligent city.* [Online] Available: https://www.researchgate.net/project/i-city-intelligent-city. Accessed on: Sep. 01 2017.

[3] Hochschule für Technik Stuttgart, *Kick-off_i_city_inkl_Partner.* Project file. Stuttgart.

[4] A. Ott, "Konzeption und Implementierung einer Internet-Plattform zum Testen von E-Bikebasierten Mobilitätskonzepten," Bachelorarbeit, Hochschule Ulm, Ulm, 2017.

[5] Open Geospatial Consortium, *OGC SensorThings API Part 1: Sensing; 15-078r6.* [Online] Available: http://docs.opengeospatial.org/is/15-078r6/15-078r6.html. Accessed on: Nov. 14 2017.

[6] S. Mosser, I. Logre, N. Ferry, and P. Collet, *From Sensors to Visualization Dashboards: Need for Language Composition, Globalization of Modeling Languages workshop (GeMOC'13).* [Online] Available: https://hal.archives-ouvertes.fr/hal-01322550.

[7] L. Kohn, "Erfassung emotionalen Erlebens von E-Bike Fahrenden mittels physiologischer Marker: eine Validierungsstudie im Forschungsprojekt „i_city" der Hochschule für Technik Stuttgart in Kooperation mit Daimler TSS," Bachelor Thesis, Wirtschaftspsychologie, Hochschule für Technik Stuttgart, Stuttgart, 2017.

[8] M. A. Jazayeri, S. H. L. Liang, and C.-Y. Huang, "Implementation and Evaluation of Four Interoperable Open Standards for the Internet of Things," (eng), *Sensors (Basel, Switzerland)*, vol. 15, no. 9, pp. 24343–24373, 2015.

[9] S. Liang, *Amazon IoT and the candidate OGC SensorThings API Standard | OGC.* [Online] Available: http://www.opengeospatial.org/blog/2315. Accessed on: Sep. 25 2017.

[10] J. Jo and I. Jang, "Applying sensor web enablement to retrieve and visualize sensor observations across the Web," in *"Towards smarter hyper-connected world": ICTC 2016 : International Conference on ICT Convergence 2016 : October 19-21, 2016, Ramada Plaza Hotel, Jeju Island, Korea*, Jeju, 2016, pp. 852–854.

[11] Open Geospatial Consortium, *Sensor Web Enablement (SWE);.* [Online] Available: http://www.opengeospatial.org/ogc/markets-technologies/swe. Accessed on: Jan. 25 2018.

[12] openweathermap, *openweathermap, Weather in Stuttgart, DE.* [Online] Available: http://openweathermap.org/city/2825297. Accessed on: Jan. 26 2018.

[13] N. Chen, L. Di, G. Yu, and J. Gong, "Geo-processing workflow driven wildfire hot pixel detection under sensor web environment," *Computers & Geosciences*, vol. 36, no. 3, pp. 362–372, 2010.

[14] 52° North Initiative for Geospatial Open Source Software GmbH, *52North/SOS: SOS 4.4.2 Release.* [Online] Available: https://github.com/52North/SOS/releases. Accessed on: Nov. 26 2017.

[15] W. Zhu, A. Simons, A. Nichersu, and S. Wursthorn, *Integration of CityGML and Air Quality Spatio-Temporal Data Series via OGC SOS: In Geospatial Sensor Webs Conference 2016.*

[16] I. Sofos, V. Vescoukis, A. Gkegkas, and E. Tsilimantou, "Applying OGC Standards to Develop a Land Surveying Measurement Model," *IJGI*, vol. 6, no. 3, p. 67, 2017.

[17] E. Partescano, A. Brosich, M. Lipizer, V. Cardin, and A. Giorgetti, "From heterogeneous marine sensors to sensor web: (near) real-time open data access adopting OGC sensor web enablement standards," *Open geospatial data, softw. stand.*, vol. 2, no. 1, p. 293, 2017.

[18] T. Slawecki, D. Young, B. Dean, B. Bergenroth, and K. Sparks, "Pilot implementation of the US EPA interoperable watershed network," *Open geospatial data, softw. stand.*, vol. 2, no. 1, p. 5291, 2017.

[19] WSV, *PEGELONLINE: Gewässerkundliches Informationssystem der Wasserstraßen- und Schifffahrtsverwaltung.* [Online] Available: http://www.pegelonline.wsv.de/gast/start. Accessed on: Nov. 22 2017.

[20] C. A. Henson, J. K. Pschorr, A. P. Sheth, and K. Thirunarayan, *SemSOS: Semantic Sensor Observation Service.* Piscataway, NJ: IEEE, 2009.

[21] S. Cox, *Observations and Measurements - XML Implementation 2.0; OGC 10-025r1.* [Online] Available: http://www.opengis.net/doc/IS/OMXML/2.0. Accessed on: Nov. 14 2017.

[22] A. Hussain and W. Wu, "Sustainable Interoperability and Data Integration for the IoT-Based Information Systems,"

[23] B. Xu *et al.*, "Ubiquitous Data Accessing Method in IoT-Based Information System for Emergency Medical Services," *IEEE Trans. Ind. Inf.*, vol. 10, no. 2, pp. 1578–1586, 2014.

[24] J. v. d. Broecke, L. Carton, M. Grothe, H. Volten, and R. Kieboom, "Smart Emission: Building a Spatial Data Infrastructure for an Environmental Citizen Sensor Network,"

[25] P. Jensen and J. Gitahi, *Visualization of Time Series in Cesium.* [Online] Available: http://gisstudio.hft-stuttgart.de/cesium.html. Accessed on: Sep. 22 2017.

[26] SensorUp Inc., *Visualizing Vancouver Water Quality Stations Data: SensorUp OGC SensorThings API Developer Centre.* [Online] Available: https://www.sensorup.com/blog/2017/12/05/visualizing-vancouver-water-quality-stations-data/. Accessed on: Dec. 12 2017.

[27] SensorUp Inc., *SensorThings API, Getting Started with Locations and Mapping.* [Online] Available: http://developers.sensorup.com/tutorials/map/. Accessed on: Dec. 12 2017.

[28] V. Agafonkin, *Leaflet, an open-source JavaScript library for interactive maps: Leaflet 1.3.1.* [Online] Available: http://leafletjs.com/. Accessed on: Dec. 12 2017.

[29] Shift Transit Inc, *BIKE SHARE TORONTO: EXPLORE THE CITY ON TWO WHEELS.* [Online] Available: https://bikesharetoronto.com/. Accessed on: Oct. 11 2017.

[30] Hochschule für Technik Stuttgart, "Geovisualisierung Projektarbeiten: Abschlussbericht WS 2016/2017," 2016.

[31] SensorUp Inc., *Comparison Between OGC Sensor Observation Service And SensorThings API | SensorUp OGC SensorThings API Developer Centre.* [Online] Available: http://developers.sensorup.com/webinars/Comparison-Between-OGC-Sensor-Observation-Service-And-SensorThings-API/. Accessed on: Oct. 01 2017.

[32] S. Liang, T. Khalafbeigi, and C. Y. Huang, *OGC SensorThings API: The official web site of the OGC SensorThings API standard specification.* [Online] Available: https://github.com/opengeospatial/sensorthings. Accessed on: Nov. 02 2017.

[33] SensorUp Inc., *Comparison between OGC Sensor Observation Service and SensorThings API.* [Online] Available: https://www.slideshare.net/steve.liang/comparison-between-ogc-sensor-observation-service-and-sensorthings-api. Accessed on: Dec. 02 2017.

[34] M. Storz and H. Dastageeri, "Usage of Advanced Sensor Data Delivery Service (ASDDS)", personal communication, Oct. 2017.

[35] S. Liang, *SensorThings SWG | OGC.* [Online] Available: http://www.opengeospatial.org/projects/groups/sweiotswg. Accessed on: Dec. 15 2017.

[36] SensorUp Inc., *OGC SensorThings API Documentation | SensorUp OGC SensorThings API Developer Centre.* [Online] Available: http://developers.sensorup.com/docs/. Accessed on: Dec. 22 2017.

[37] GeoCENS, *GeoCENS: Sensor Web Made Easy.* [Online] Available: http://www.geocens.ca/. Accessed on: Oct. 11 2017.

[38] I. Simonis, *OGC Sensor Web Enablement Architecture; 06-021r4.* [Online] Available: https://portal.opengeospatial.org/files/?artifact_id=29405. Accessed on: Nov. 29 2017.

[39] SensorUp Inc., *Build Interoperable Smart City Applications Effortlessly: SensorUp OGC SensorThings API Developer Centre.* [Online] Available: http://developers.sensorup.com/webinars/Build-Interoperable-Smart-City-Applications-Effortlessly/. Accessed on: Dec. 12 2017.

[40] Wikipedia, *Sensor Observation Service - Wikipedia.* [Online] Available: https://en.wikipedia.org/w/index.php?oldid=717940735. Accessed on: Nov. 23 2017.

[41] Open Geospatial Consortium, *Sensor Observation Service | OGC.* [Online] Available: http://www.opengeospatial.org/standards/sos. Accessed on: Nov. 25 2017.

[42] Open Geospatial Consortium, *OGC 12-006; OGC Sensor Observation Service Interface Standard.* [Online] Available: http://www.opengis.net/doc/IS/SOS/2.0. Accessed on: Oct. 22 2017.

[43] OGC Network, *How to model your observation data in SOS 2.0? | OGC Network.* [Online] Available: http://www.ogcnetwork.net/sos_2_0/tutorial/om. Accessed on: Jan. 25 2018.

[44] EUROPÄISCHES DATENPORTAL, *PEGELONLINE Sensor Observation Service - European Data Portal.* [Online] Available: https://www.europeandataportal.eu/data/de/dataset/pegelonline/resource/65835f97-6020-4ebe-9e37-39d8d83f1ee0?inner_span=True. Accessed on: Nov. 22 2017.

[45] 52° North Initiative for Geospatial Open Source Software GmbH, *PegelOnline SOS TestClient; Version 2.* [Online] Available: http://sensorweb.demo.52north.org/PegelOnlineSOSv2.1/. Accessed on: Oct. 15 2017.

[46] Highcharts, *Highcharts Javascript Charting Library - Highcharts: EVERYBODY'S FAVORITE CHARTING LIBRARY.* [Online] Available: https://www.highcharts.com/products/highcharts/. Accessed on: Dec. 25 2017.

[47] Yammer Inc., *Dropwizard: Production-ready, out of the box.* [Online] Available: http://www.dropwizard.io/1.2.2/docs/. Accessed on: Dec. 20 2017.

[48] The PostgreSQL Global Development Group, *PostgreSQL: The world's most advanced open source database.* [Online] Available: https://www.postgresql.org/. Accessed on: Oct. 01 2017.

[49] movisens GmbH, *EdaMove 3: EDA and Physical Activity Sensor.* [Online] Available: https://www.movisens.com/en/products/edamove-3/. Accessed on: Jan. 03 2018.

[50] movisens GmbH, *EcgMove 3: ECG and Activity Sensor.* [Online] Available: https://www.movisens.com/en/products/ecg-and-activity-sensor-ecgmove-3/. Accessed on: Jan. 03 2018.

[51] Garmin Ltd., *fenix 5X, Fitness GPS watch.* [Online] Available: https://buy.garmin.com/en-US/US/p/560327. Accessed on: Dec. 01 2017.

[52] Daimler TSS GmbH, *Car IT & Mobility – A Daimler TSS product line.* [Online] Available: https://www.daimler-tss.com/en/strategic-fields/car-it-mobility/. Accessed on: Oct. 11 2017.

[53] car2go Deutschland GmbH, *car2go: STUTTGART IS PROUD TO SHARE.* [Online] Available: https://www.car2go.com/DE/en/stuttgart/. Accessed on: Jan. 08 2018.

[54] M. Krämer and R. Gutbell, "A case study on 3D geospatial applications in the web using state-of-the-art WebGL frameworks," in *Proceedings of the 20th International Conference on 3D Web Technology - Web3D '15*, Heraklion, Crete, Greece, 2015, pp. 189–197.

[55] Cesium Consortium, *Cycling the Alps migrates to Cesium | cesium.com.* [Online] Available: https://cesium.com/blog/2015/02/05/cycling-the-alps-migrates-to-cesium/. Accessed on: Nov. 25 2017.

[56] S. Hunter, *Cesium Language (CZML) Guide: AnalyticalGraphicsInc/czml-writer.* [Online] Available: https://github.com/AnalyticalGraphicsInc/czml-writer/wiki/CZML-Guide. Accessed on: Dec. 01 2017.

[57] Cycling the Alps, *Cycling the Alps.* [Online] Available: http://www.cyclingthealps.com/. Accessed on: Dec. 05 2017.

[58] Cesium Consortium, *Cesium | cesium.com: An open-source JavaScript library for world-class 3D globes and maps.* [Online] Available: https://cesiumjs.org/. Accessed on: Dec. 03 2017.

[59] Cesium Consortium, *STK World Terrain | cesiumjs.org.* [Online] Available: https://cesiumjs.org/data-and-assets/terrain/stk-world-terrain/. Accessed on: Jan. 05 2018.

[60] Cesium Consortium, *Cesium Sandcastle: Terrain Showcase.* [Online] Available: https://cesiumjs.org/Cesium/Build/Apps/Sandcastle/index.html?src=Terrain.html&label=Showcases. Accessed on: Dec. 14 2017.

[61] S. Hunter, *Path: AnalyticalGraphicsInc/czml-writer.* [Online] Available: https://github.com/AnalyticalGraphicsInc/czml-writer/wiki/Path. Accessed on: Nov. 27 2017.

[62] Cesium Consortium, *Cesium Sandcastle: CZML Path.* [Online] Available: https://cesiumjs.org/Cesium/Build/Apps/Sandcastle/index.html?src=CZML%20Path.html&label=CZML. Accessed on: Dec. 14 2017.

[63] S. Hunter, *InterpolatableProperty: AnalyticalGraphicsInc/czml-writer.* [Online] Available: https://github.com/AnalyticalGraphicsInc/czml-writer/wiki/InterpolatableProperty. Accessed on: Nov. 27 2017.

[64] Landeshauptstadt Stuttgart, *Stuttgart3D.* [Online] Available: http://www.stuttgart.de/3d. Accessed on: Sep. 01 2017.

[65] TU Delft, *CityGML.* [Online] Available: https://www.citygml.org/. Accessed on: Jan. 03 2018.

[66] Safe Software Inc, *FME Desktop: Data Integration & Productivity.* [Online] Available: https://www.safe.com/fme/fme-desktop/. Accessed on: Jan. 03 2018.

[67] Esri Inc., *ArcGIS Online Basemaps.* [Online] Available: http://www.arcgis.com/home/group.html?id=702026e41f6641fb85da88efe79dc166#overview. Accessed on: Jan. 03 2018.

[68] Fraunhofer Institute for Computer Graphics Research IGD, *GeoRocket: GeoToolbox 1.0.2.* [Online] Available: https://georocket.io/products/. Accessed on: Jan. 03 2018.

[69] Technische Universität München, *3D City Database Software.* [Online] Available: https://www.3dcitydb.org/3dcitydb/software. Accessed on: Jan. 03 2018.

[70] The Khronos Group Inc, *Khronos Finalizes glTF 1.0 Specification.* [Online] Available: https://www.khronos.org/news/press/khronos-finalizes-gltf-1.0-specification. Accessed on: Jan. 01 2018.

[71] Wikipedia, *glTF - Wikipedia.* [Online] Available: https://en.wikipedia.org/w/index.php?oldid=822173051. Accessed on: Jan. 25 2018.

[72] Technische Universität München, *3DCityDB-Web-Map-Client.* [Online] Available: https://github.com/3dcitydb/3dcitydb-web-map. Accessed on: Jan. 03 2018.

[73] Cesium Consortium, *Introducing 3D Tiles | cesium.com.* [Online] Available: https://cesium.com/blog/2015/08/10/introducing-3d-tiles/. Accessed on: Dec. 30 2017.

[74] H. v. d. Schaaf and M. Jacoby, *FraunhoferIOSB/SensorThingsServer: A Server implementation of the OGC SensorThings API.* [Online] Available: https://github.com/FraunhoferIOSB/SensorThingsServer. Accessed on: Dec. 19 2017.

[75] Apache Software Foundation, *Tomcat Maven Plugin.* [Online] Available: http://tomcat.apache.org/maven-plugin.html. Accessed on: Nov. 19 2017.

[76] T. Ornelas, *json-viewer: Chrome extension for printing JSON and JSONP.* [Online] Available: https://github.com/tulios/json-viewer. Accessed on: Dec. 24 2017.

[77] Postdot Technologies, Inc., *Postman.* [Online] Available: https://www.getpostman.com/postman. Accessed on: Dec. 23 2017.

[78] 52° North Initiative for Geospatial Open Source Software GmbH., *Sensor Web Technology for Monitoring the Environment.* [Online] Available: https://52north.org/. Accessed on: Jan. 26 2018.

[79] 52° North Initiative for Geospatial Open Source Software GmbH, *Sensor Observation Service Developer Guide: SOS 4.x Contributor Guide.* [Online] Available: https://wiki.52north.org/SensorWeb/SensorObservationServiceDeveloperGuide#Database. Accessed on: Nov. 20 2017.

[80] Open Geospatial Consortium, *Geographic information — Observations and measurements; OGC 10-004r3.* [Online] Available: http://www.opengis.net/doc/is/om/2.0. Accessed on: Nov. 28 2017.

[81] Canonical, *Ubuntu 17.10.* [Online] Available: https://www.ubuntu.com/desktop/1710. Accessed on: Oct. 01 2017.

[82] Oracle, *Oracle VM VirtualBox.* [Online] Available: https://www.virtualbox.org/wiki/Downloads. Accessed on: Oct. 01 2017.

[83] SmartBear Software, *Swagger, World's Most Popular API Framework.* [Online] Available: https://swagger.io/. Accessed on: Oct. 29 2017.

[84] Garmin Ltd., *Garmin Connect.* [Online] Available: https://connect.garmin.com/en-US/status. Accessed on: Dec. 01 2017.

[85] *JSON to CSV Converter Online.* [Online] Available: https://json-csv.com/. Accessed on: Nov. 01 2017.

[86] M. Zeiss, *json2csv: Converts json into csv with column titles and proper line endings.* [Online] Available: https://www.npmjs.com/package/json2csv. Accessed on: Dec. 04 2017.

[87] openweathermap, *Hourly historical data for cities.* [Online] Available: http://openweathermap.org/history. Accessed on: Jan. 26 2018.

[88] FreeVectors.net, *Biker Vector Image.* [Online] Available: http://www.freevectors.net/details/Biker+Vector+Image. Accessed on: Jan. 04 2018.

[89] m. rogers and S. Velichkov, *request.* [Online] Available: https://www.npmjs.com/package/request. Accessed on: Dec. 24 2017.

[90] W. David, *csv-parse: CSV parsing implementing the Node.js `stream.Transform` API.* [Online] Available: https://github.com/adaltas/node-csv-parse. Accessed on: Dec. 04 2017.

[91] J. P. Sirois and Z. Hall, *Lodash: A modern JavaScript utility library delivering modularity, performance & extras.* [Online] Available: https://lodash.com/. Accessed on: Dec. 04 2017.

[92] M. Conrad, *xml-parse.* [Online] Available: https://www.npmjs.com/package/xml-parse. Accessed on: Dec. 04 2017.

[93] Technische Universität München, *3D City Database Importer/Exporter.* [Online] Available: https://github.com/3dcitydb/importer-exporter. Accessed on: Jan. 03 2018.

[94] T. H. kolbe *et al.*, "3D City Database for CityGML: version 3.3.0 documentation," 2016.

[95] Cesium Consortium, *Quadtree Cheatsheet | cesium.com.* [Online] Available: https://cesium.com/blog/2015/04/07/quadtree-cheatseet/. Accessed on: Dec. 05 2017.

[96] Safe Software Inc, *Cesium 3D Tiles Writer.* [Online] Available: https://docs.safe.com/fme/html/FME_Desktop_Documentation/FME_ReadersWriters/cesium3dtiles/cesium3dtiles.htm. Accessed on: Nov. 28 2017.

[97] N. j. Foundation, *Node.js.* [Online] Available: https://nodejs.org/en/. Accessed on: Jan. 13 2018.

[98] Geofabrik GmbH and OpenStreetMap Contributors, *OpenStreetMap Data Extracts.* [Online] Available: https://download.geofabrik.de/. Accessed on: Nov. 01 2017.

[99] Mapbox, *Maki Icons: An icon set made for map designers.* [Online] Available: https://www.mapbox.com/maki-icons/. Accessed on: Nov. 16 2017.

[100] The jQuery Foundation, *jQuery: write less, do more.* [Online] Available: http://jquery.com/. Accessed on: Nov. 29 2017.

[101] W. Vega, *jQuery Timepicker.* [Online] Available: http://timepicker.co/. Accessed on: Nov. 09 2017.

[102] *jStat: JavaScript Statistical Library.* [Online] Available: https://github.com/jstat/jstat. Accessed on: Dec. 09 2017.

[103] C. Bormann, A. P. Castellani, and Z. Shelby, "CoAP: An Application Protocol for Billions of Tiny Internet Nodes," *IEEE Internet Comput.*, vol. 16, no. 2, pp. 62–67, 2012.

[104] alertra, *Role of Server Response Time in Website Performance.* [Online] Available: https://www.alertra.com/blog/2012/role-server-response-time-website-performance. Accessed on: Jan. 29 2018.

[105] G. Getz, "Cesium update in 2018", cesium-developer group, Jan. 2018.

[106] Open Geospatial Consortium, *OGC Announces New 3D Portrayal Service Standard | OGC.* [Online] Available: http://www.opengeospatial.org/pressroom/pressreleases/2651. Accessed on: Dec. 13 2017.

[107] Hochschule für Technik Stuttgart, *SunnySide: GeoVisualisierung.* [Online] Available: https://www.hft-stuttgart.de/Aktuell/Nachrichten/j2018/m0118/geovisualisierung_sunny_side/de. Accessed on: Feb. 12 2018.

Appendix

Appendix A. Sensors Data

Appendix A - 1. The example of the Observation data from Garmin Smart Watch fēnix 5X (GPX-file) collected on 2017-11-21

Table 10: Showing the first 50 rows of the Observation data from Garmin Smart Watch fēnix 5X (GPX-file) collected on 2017-11-21 by converting the GPX from XML base to CSV base

Latitude (degree)	Longitude (degree)	Altitude (m)	Time	Temperature(°C)	Heartrate (bpm)
48.78218679	9.177650632	212.6000061	2017-11-21T14:40:04.000Z	32	85
48.78215703	9.177610483	212.8000031	2017-11-21T14:40:07.000Z	32	82
48.78200423	9.177916674	212.8000031	2017-11-21T14:40:31.000Z	32	79
48.78198646	9.177944167	212.8000031	2017-11-21T14:40:33.000Z	32	75
48.78198101	9.177982891	212.8000031	2017-11-21T14:40:35.000Z	32	72
48.78199778	9.178048186	212.6000061	2017-11-21T14:40:42.000Z	32	75
48.78190692	9.178200737	212.3999939	2017-11-21T14:40:46.000Z	32	75
48.78188596	9.178184392	212.8000031	2017-11-21T14:40:49.000Z	33	78
48.78182888	9.178231917	212.6000061	2017-11-21T14:40:53.000Z	33	81
48.78196517	9.178235354	213	2017-11-21T14:41:41.000Z	33	78
48.78201957	9.178108368	213	2017-11-21T14:41:48.000Z	33	82
48.78202292	9.17810577	213	2017-11-21T14:41:49.000Z	33	86
48.78202686	9.178101579	213	2017-11-21T14:41:50.000Z	33	89
48.7820303	9.178098394	213	2017-11-21T14:41:51.000Z	33	89
48.7820831	9.17795632	213	2017-11-21T14:41:56.000Z	33	90
48.7820914	9.177912734	213	2017-11-21T14:41:58.000Z	33	85
48.78209635	9.177900916	213	2017-11-21T14:41:59.000Z	33	80
48.7820422	9.177999236	213	2017-11-21T14:42:00.000Z	33	77
48.78202334	9.178141393	212.6000061	2017-11-21T14:42:15.000Z	33	74
48.78200775	9.178224122	212.8000031	2017-11-21T14:42:31.000Z	33	78
48.7819681	9.178117001	212.8000031	2017-11-21T14:42:42.000Z	33	81

Latitude (degree)	Longitude (degree)	Altitude (m)	Time	Temperature(°C)	Heartrate (bpm)
48.78196551	9.178107949	212.8000031	2017-11-21T14:42:47.000Z	33	84
48.78194874	9.178047264	212.8000031	2017-11-21T14:43:02.000Z	33	81
48.78194505	9.178039888	212.6000061	2017-11-21T14:43:05.000Z	33	78
48.7819469	9.178032344	212.6000061	2017-11-21T14:43:07.000Z	33	75
48.78195008	9.17799647	212.8000031	2017-11-21T14:43:10.000Z	33	78
48.78197397	9.178060843	212.8000031	2017-11-21T14:43:16.000Z	33	81
48.78201177	9.178091856	212.8000031	2017-11-21T14:43:19.000Z	33	84
48.78203834	9.178074254	212.8000031	2017-11-21T14:43:23.000Z	33	87
48.78208436	9.17794928	212.8000031	2017-11-21T14:43:35.000Z	33	90
48.78209777	9.177928409	212.8000031	2017-11-21T14:43:39.000Z	33	93
48.78211261	9.177911812	212.8000031	2017-11-21T14:43:42.000Z	33	96
48.78213373	9.177883482	212.8000031	2017-11-21T14:43:45.000Z	33	99
48.78216793	9.177844087	212.8000031	2017-11-21T14:43:50.000Z	33	93
48.78218696	9.177842913	212.8000031	2017-11-21T14:43:51.000Z	33	88
48.7822204	9.177846434	212.8000031	2017-11-21T14:43:52.000Z	33	84
48.78224689	9.177846517	212.8000031	2017-11-21T14:43:53.000Z	33	84
48.7822862	9.178101746	212.8000031	2017-11-21T14:44:06.000Z	33	81
48.78209786	9.178376673	212.8000031	2017-11-21T14:44:13.000Z	33	79
48.78195008	9.178581191	212.8000031	2017-11-21T14:44:17.000Z	33	82
48.78192142	9.178624442	212.8000031	2017-11-21T14:44:18.000Z	33	85
48.78184246	9.178733155	212.8000031	2017-11-21T14:44:20.000Z	33	89
48.78165001	9.179040184	212.8000031	2017-11-21T14:44:28.000Z	33	86
48.78157994	9.17911537	212.8000031	2017-11-21T14:44:31.000Z	33	83
48.78139696	9.179515019	212.8000031	2017-11-21T14:44:43.000Z	33	87
48.78121231	9.179993877	213.1999969	2017-11-21T14:44:53.000Z	33	91
48.78122882	9.17998801	213	2017-11-21T14:44:55.000Z	33	94
48.78123888	9.179988597	213	2017-11-21T14:44:57.000Z	33	98
48.78215846	9.178177519	213.6000061	2017-11-21T14:45:32.000Z	33	95
...	...	...	...	...	...

Appendix A - 2. The example of the Observation data from Garmin Smart Watch fēnix 5X (TCX-file) collected on 2017-11-21

Table 11: Showing the first 50 rows of the Observation data from Garmin Smart Watch fēnix 5X (TCX-file) collected on 2017-11-21 by converting the TCX from XML base to CSV base

Time	Distance (m)	Heartrate (bpm)	Speed (km/h)	Latitude (degree)	Longitude (degree)	Altitude (m)
2017-11-21T14:40:04.000Z	5.789999962	85	0	48.78218679	9.177650632	212.6000061
2017-11-21T14:40:07.000Z	10.18999958	82	0.504000008	48.78215703	9.177610483	212.8000031
2017-11-21T14:40:31.000Z	37.54999924	79	0.792999983	48.78200423	9.177916674	212.8000031
2017-11-21T14:40:33.000Z	40.22000122	75	0.764999986	48.78198646	9.177944167	212.8000031
2017-11-21T14:40:35.000Z	43.08000183	72	0.755999982	48.78198101	9.177982891	212.8000031
2017-11-21T14:40:42.000Z	46.91999817	75	0.560000002	48.78199778	9.178048186	212.6000061
2017-11-21T14:40:46.000Z	61.81000137	75	0.904999971	48.78190692	9.178200737	212.3999939
2017-11-21T14:40:49.000Z	65.37999725	78	0.82099998	48.78188596	9.178184392	212.8000031
2017-11-21T14:40:53.000Z	72.52999878	81	0.829999983	48.78182888	9.178231917	212.6000061
2017-11-21T14:41:41.000Z	95.25	78	0	48.78196517	9.178235354	213
2017-11-21T14:41:48.000Z	106.4199982	82	0.680999994	48.78201957	9.178108368	213
2017-11-21T14:41:49.000Z	106.8199997	86	0.671999991	48.78202292	9.17810577	213
2017-11-21T14:41:50.000Z	107.3600006	89	0.662	48.78202686	9.178101579	213
2017-11-21T14:41:51.000Z	107.8000031	89	0.662	48.7820303	9.178098394	213
2017-11-21T14:41:56.000Z	119.4100037	90	0.896000028	48.7820831	9.17795632	213
2017-11-21T14:41:58.000Z	122.5400009	85	0.941999972	48.7820914	9.177912734	213
2017-11-21T14:41:59.000Z	123.5599976	80	0.961000025	48.78209635	9.177900916	213
2017-11-21T14:42:00.000Z	123.5599976	77	0.662	48.7820422	9.177999236	213
2017-11-21T14:42:15.000Z	126.3600006	74	0	48.78202334	9.178141393	212.6000061
2017-11-21T14:42:31.000Z	134.5200043	78	0	48.78200775	9.178224122	212.8000031
2017-11-21T14:42:42.000Z	143.9400024	81	0	48.7819681	9.178117001	212.8000031
2017-11-21T14:42:47.000Z	144.6699982	84	0	48.78196551	9.178107949	212.8000031
2017-11-21T14:43:02.000Z	149.6000061	81	0	48.78194874	9.178047264	212.8000031
2017-11-21T14:43:05.000Z	150.2700043	78	0	48.78194505	9.178039888	212.6000061
2017-11-21T14:43:07.000Z	150.7100067	75	0	48.7819469	9.178032344	212.6000061

Time	Distance (m)	Heartrate (bpm)	Speed (km/h)	Latitude (degree)	Longitude (degree)	Altitude (m)
2017-11-21T14:43:10.000Z	153.1799927	78	0	48.78195008	9.17799647	212.8000031
2017-11-21T14:43:16.000Z	154.1999969	81	0	48.78197397	9.178060843	212.8000031
2017-11-21T14:43:19.000Z	158.1399994	84	0	48.78201177	9.178091856	212.8000031
2017-11-21T14:43:23.000Z	161.1600037	87	0	48.78203834	9.178074254	212.8000031
2017-11-21T14:43:35.000Z	175.2200012	90	0.495000005	48.78208436	9.17794928	212.8000031
2017-11-21T14:43:39.000Z	177.3399963	93	0.495000005	48.78209777	9.177928409	212.8000031
2017-11-21T14:43:42.000Z	179.3399963	96	0.523000002	48.78211261	9.177911812	212.8000031
2017-11-21T14:43:45.000Z	182.4499969	99	0.569000006	48.78213373	9.177883482	212.8000031
2017-11-21T14:43:50.000Z	187.0700073	93	0.643999994	48.78216793	9.177844087	212.8000031
2017-11-21T14:43:51.000Z	188.4499969	88	0.671999991	48.78218696	9.177842913	212.8000031
2017-11-21T14:43:52.000Z	190.9499969	84	0.727999985	48.7822204	9.177846434	212.8000031
2017-11-21T14:43:53.000Z	193.2299957	84	0.783999979	48.78224689	9.177846517	212.8000031
2017-11-21T14:44:06.000Z	199.3999939	81	0.578999996	48.7822862	9.178101746	212.8000031
2017-11-21T14:44:13.000Z	219.3099976	79	0.896000028	48.78209786	9.178376673	212.8000031
2017-11-21T14:44:17.000Z	241.3800049	82	1.45599997	48.78195008	9.178581191	212.8000031
2017-11-21T14:44:18.000Z	245.8800049	85	1.539999962	48.78192142	9.178624442	212.8000031
2017-11-21T14:44:20.000Z	257.75	89	1.819000006	48.78184246	9.178733155	212.8000031
2017-11-21T14:44:28.000Z	289.0700073	86	2.183000088	48.78165001	9.179040184	212.8000031
2017-11-21T14:44:31.000Z	298.5799866	83	2.249000072	48.78157994	9.17911537	212.8000031
2017-11-21T14:44:43.000Z	334.5899963	87	2.407000065	48.78139696	9.179515019	212.8000031
2017-11-21T14:44:53.000Z	375.3500061	91	2.780999899	48.78121231	9.179993877	213.1999969
2017-11-21T14:44:55.000Z	376.0400085	94	2.594000101	48.78122882	9.17998801	213
2017-11-21T14:44:57.000Z	376.3900146	98	2.453999996	48.78123888	9.179988597	213
2017-11-21T14:44:58.000Z	376.3900146	102	2.453999996	(Null)	(Null)	213
...	...	...	...	...	...	...

Appendix A - 3. Smart Electric Bike JSON API response body structure

```
                'vin', 'change.geo.altitude',
                'change.geo.latitude', 'change.geo.longitude',
                'change.geo.accuracy', 'change.batteryVoltageBike',
                'change.connection.connected', 'change.mileage',
                'change.pedalForce', 'change.speed',
                'change.motorSupport.min', 'change.motorSupport.avg',
                'change.motorSupport.max', 'change.motorSupport.count',
                'change.motorSupport.firstValTime', 'change.motorSupport.timeSpan',
                'change.motorSupport.lastVal', 'change.light',
                'change.fuelLevel', 'uuid',
                'status.allDoorsClosed', 'status.fuelLevel',
                'status.charging', 'status.ignitionOn',
                'status.geo.altitude', 'status.geo.latitude',
                'status.geo.accuracy', 'status.geo.longitude',
                'status.powerState', 'status.light',
                'status.connection.connected', 'status.connection.since',
                'status.locked', 'status.batteryVoltageBike',
                'status.mileage', 'status.batteryLevel',
                'timestamp'
```

Figure 81: Smart Electric Bike JSON API response body structure

Appendix A - 4. Example of the Smart E-bike log data retrieve by FTP solution


```
{"change":{"batteryVoltageBike":51610},"vin":"eBike20131126002f","uuid":"92152855-4b58-4f13-
9f94-
318a165e97ea","status":{"allDoorsClosed":true,"fuelLevel":99,"charging":true,"ignitionOn":fals
e,"geo":{"latitude":48.42279639,"accuracy":3,"longitude":9.94217799},"powerState":"ON","light"
:false,"connection":{"connected":true,"since":"Nov 21, 2017 12:01:20
AM"},"vin":"eBike20131126002f","locked":false,"batteryVoltageBike":51610,"mileage":70788,"batt
eryLevel":5.653644},"timestamp":"2017-11-21_02:01:17.379"}

{"change":{"batteryVoltageBike":53351},"vin":"eBike20131127000a","uuid":"8732be68-2d66-4665-
b21b-
b90b5ad35ffa","status":{"allDoorsClosed":true,"fuelLevel":100,"charging":true,"ignitionOn":fal
se,"geo":{"latitude":48.42233424,"accuracy":4,"longitude":9.94182737},"powerState":"ON","light
":false,"connection":{"connected":true,"since":"Nov 21, 2017 12:07:30
AM"},"vin":"eBike20131127000a","locked":false,"batteryVoltageBike":53351,"mileage":4092,"batte
ryLevel":5.403541},"timestamp":"2017-11-21_02:07:28.162"}

{"change":{"connection":{"connected":false}},"vin":"eBike20131126003c","uuid":"2fc21b2f-b6e5-
43df-a31e-
642347e598e3","status":{"allDoorsClosed":true,"fuelLevel":43,"charging":true,"ignitionOn":fals
e,"geo":{"latitude":48.7801678,"accuracy":3,"longitude":9.173689490000001},"powerState":"ON","
light":false,"connection":{"connected":false,"since":"Nov 21, 2017 12:03:36
AM"},"vin":"eBike20131126003c","locked":false,"batteryVoltageBike":48895,"mileage":40630,"batt
eryLevel":5.308369},"timestamp":"2017-11-21_02:02:36.750"}

{"change":{"batteryVoltageBike":51560},"vin":"eBike20131126002f","uuid":"5b31260d-e073-4954-
bbb9-
cc785f9a52f2","status":{"allDoorsClosed":true,"fuelLevel":99,"charging":true,"ignitionOn":fals
e,"geo":{"latitude":48.42279639,"accuracy":3,"longitude":9.94217799},"powerState":"ON","light"
:false,"connection":{"connected":true,"since":"Nov 21, 2017 12:11:29
AM"},"vin":"eBike20131126002f","locked":false,"batteryVoltageBike":51560,"mileage":70788,"batt
eryLevel":5.6660385},"timestamp":"2017-11-21_02:11:26.554"}
```


Figure 82: Example of the E-bike log events retrieve by FTP solution

Appendix A - 5. NodeJS program for converting the Smart E-bike log data into CSV format

```
const fs = require('fs');

var json2csv = require('json2csv');
// ==============Loading Properties==============
var file = 'events_2017-11-21'; //input log file name
//==============================================
// Loading File
var fileN = file + '.log';
var outname = `${file}_out.csv`;
var obj = (fs.readFileSync(fileN, 'utf8'));
obj = obj.replace(/\n/g, ',')
obj = '[' + obj + ']';
var obj2 = JSON.parse(obj);
console.log('file %s load successful', fileN);
function conjson() {
var fields = [  'vin',
                'change.geo.altitude', 'change.geo.latitude',
                'change.geo.longitude','change.geo.accuracy',
                'change.batteryVoltageBike','change.connection.connected',
                'change.mileage','change.pedalForce',
                'change.speed','change.motorSupport.min',
                'change.motorSupport.avg','change.motorSupport.max',
                'change.motorSupport.count','change.motorSupport.firstValTime',
                'change.motorSupport.timeSpan','change.motorSupport.lastVal',
                'change.light','change.fuelLevel',
                'vin','uuid',
                'status.allDoorsClosed', 'status.fuelLevel',
                'status.charging','status.ignitionOn',
                'status.geo.altitude','status.geo.latitude',
                'status.geo.accuracy','status.geo.longitude',
                'status.powerState','status.light',
                'status.connection.connected','status.connection.since',
                'status.locked','status.batteryVoltageBike',
                'status.mileage','status.batteryLevel',
                'timestamp'
                ];
var csv = json2csv({ data: obj2, fields: fields });
fs.writeFile(outname, csv, function(err) {
  if (err) throw err;
  console.log('file %s csv saved', outname);
});
}
try {
    conjson()
}
catch(err) {
    console.log('->  Update of json successful!\n' + err);
}
```

Figure 83: NodeJS program to convert the Smart E-bike log data to CSV

Appendix A - 6. Example of the Smart E-bike log data after converting into CSV format

```
"vin","change.geo.altitude","change.geo.latitude","change.geo.longitude","change.geo.accuracy","change.batteryVoltageBike","change.connection.connected","change.mileage","change.pedalForce","change.speed","change.motorSupport.min","change.motorSupport.avg","change.motorSupport.max","change.motorSupport.count","change.motorSupport.firstValTime","change.motorSupport.timeSpan","change.motorSupport.lastVal","change.light","change.fuelLevel","vin","uuid","status.allDoorsClosed","status.fuelLevel","status.charging","status.ignitionOn","status.geo.altitude","status.geo.latitude","status.geo.accuracy","status.geo.longitude","status.powerState","status.light","status.connection.connected","status.connection.since","status.locked","status.batteryVoltageBike","status.mileage","status.batteryLevel","timestamp"

"eBike20131126002f",,,,,51610,,,,,,,,,,,,,"eBike20131126002f","92152855-4b58-4f13-9f94-318a165e97ea",true,99,true,false,,48.42279639,3,9.94217799,"ON",false,true,"Nov 21, 2017 12:01:20 AM",false,51610,70788,5.653644,"2017-11-21_02:01:17.379"

"eBike20131127000a",,,,,53351,,,,,,,,,,,,,"eBike20131127000a","8732be68-2d66-4665-b21b-b90b5ad35ffa",true,100,true,false,,48.42233424,4,9.94182737,"ON",false,true,"Nov 21, 2017 12:07:30 AM",false,53351,4092,5.403541,"2017-11-21_02:07:28.162"
```

Figure 84: The first three lines of the CSV output file by converting the E-bike log events

Appendix A - 7. OpenWeatherMap Weather parameters in API respond for hourly historical data for cities [87]

Table 12: Weather parameters in API respond for hourly historical data for cities [87]

Parameters	Parameter Description
coord.lon	City geo location, longitude
coord.lat	City geo location, latitude
weather.id	Weather condition id
weather.main	Group of weather parameters (Rain, Snow, Extreme etc.)
weather.description	Weather condition within the group
weather.icon	Weather icon id
main.temp	Temperature. Unit Default: Kelvin, Metric: Celsius, Imperial: Fahrenheit.
main.pressure	Atmospheric pressure (on the sea level, if there is no sea_level or grnd_level data), hPa
main.humidity	Humidity, %
main.temp_min	Minimum temperature at the moment. This is deviation from temperature that is possible for large cities and megalopolises geographically expanded. Unit Default: Kelvin, Metric: Celsius, Imperial: Fahrenheit.
main.temp_max	Maximum temperature at the moment. This is deviation from temperature that is possible for large cities and megalopolises geographically expanded. Unit Default: Kelvin, Metric: Celsius, Imperial: Fahrenheit.
main.sea_level	Atmospheric pressure on the sea level, hPa
main.grnd_level	Atmospheric pressure on the ground level, hPa

Parameters	Parameter Description
wind.speed	Wind speed. Unit Default: meter/sec, Metric: meter/sec, Imperial: miles/hour.
wind.deg	Wind direction, degrees (meteorological)
clouds.all	Cloudiness, %
rain.3h	Rain volume for the last 3 hours
snow.3h	Snow volume for the last 3 hours
dt	Time of data calculation, unix, UTC
id	City ID

Appendix A - 8. Movisens Sensor Data parameters

Table 13: EdaMove 3 data parameter [49]

Parameter name	Unit	Parameter Definition
Time rel	[s]	Relative time from start of measurements in seconds
Day rel	[d]	Number of days from start of measurement
Time rel	[hh:mm:ss]	Relative time from start if measurement
Date abs	[yyyy-mm-dd]	Absolute date
Time abs	[hh:mm:ss]	Absolute time
Altitude	[m]	Altitude from barometer
EdaArousalMean	[-]	Mean of EDA arousal in output interval
EdaSclMean	[uS]	Skin conductance level
EdaScrAmplitudesMean	[uS]	Mean of SCR Amplitudes in output interval
EdaScrCount	[-]	Number of SCRs in output interval
EdaScrEnergiesMean	[uSs]	Mean of SCR Energy in output interval
EdaScrHalfRecoveryTimesMean	[uSs]	Mean of SCR Half Recovery Times in output interval
EdaScrRiseTimesMean	[s]	Mean of SCR Rise Times in output interval
TempMean	[-]	Mean temperature in output interval
VerticalSpeed	[m/s]	Vertical speed, calculated from barometer
stateOfCharge	[%]	State of Charge

Table 14: Movisens EcgMove 3 sensor data parameter [50]

Parameter name	Unit	Parameter Definition
Time rel	[s]	Relative time from start of measurements in seconds
Day rel	[d]	Number of days from start of measurement
Time rel	[hh:mm:ss]	Relative time from start if measurement
Date abs	[yyyy-mm-dd]	Absolute date
Time abs	[hh:mm:ss]	Absolute time
Altitude	[m]	Altitude from barometer
BaevskyStressIndex	[-]	Baevsky Stress Index

Parameter name	Unit	Parameter Definition
Hr	[1/min]	Mean heartrate in output interval [1/min]
HrvHf	[ms^2]	HRV parameter HF (high frequency)
HrvLf	[ms^2]	HRV parameter LF (low frequency)
HrvLfHf	[-]	HRV parameter LF/HF (low frequency to high frequency ratio)
HrvPnn50	[%]	HRV parameter pNNx
HrvRmssd	[ms]	HRV parameter RMSSD (root mean square of successive differences of NN intervals)
HrvSd1	[ms]	HRV parameter SD1 ()
HrvSd2	[ms]	HRV parameter SD2 ()
HrvSd2Sd1	[ms]	HRV parameter SD2/SD1 (SD2 to SD1 ration)
HrvSdnn	[ms]	HRV parameter SDNN (standard deviation of NN intervals)
HrvSdsd	[ms]	HRV parameter SDSD (standard deviation of successive differences of NN intervals)
TempMean	[-]	Mean temperature in output interval
VerticalSpeed	[m/s]	Vertical speed, calculated from barometer
hrvIsValid	[bool]	hrvIsValid
hrvRmssd	[ms]	hrvRmssd
movementAcceleration	[g]	movementAcceleration

Appendix B. Sensor Server Implementation

Appendix B - 1. 52° North Sensor Observation Service "InsertSensor" operation request

Service URL

http://localhost:8080/52n-sos-webapp/service

Request

POST | application/json | application/json | Permalink | Syntax


```
{
  "request": "InsertSensor",
  "service": "SOS",
  "version": "2.0.0",
  "procedureDescriptionFormat": "http://www.opengis.net/sensorML/1.0.1",
  "procedureDescription": "<sml:SensorML xmlns:swes=\"http://www.opengis.net/swes/2.0\" xmlns:sos=\"http://www.opengis.net/sos/2.0\"
xmlns:swe=\"http://www.opengis.net/swe/1.0.1\" xmlns:sml=\"http://www.opengis.net/sensorML/1.0.1\"
xmlns:gml=\"http://www.opengis.net/gml\" xmlns:xlink=\"http://www.w3.org/1999/xlink\" xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-
instance\" version=\"1.0.1\"><sml:member><sml:System><sml:identification><sml:IdentifierList><sml:identifier name=\"uniqueID\">
<sml:Term definition=\"urn:ogc:def:identifier:OGC:1.0:uniqueID\"><sml:value>http://www.52north.org/test/procedure/9</sml:value>
</sml:Term></sml:identifier><sml:identifier name=\"longName\"><sml:Term definition=\"urn:ogc:def:identifier:OGC:1.0:longName\">
<sml:value>52°North Initiative for Geospatial Open Source Software GmbH (http://52north.org)</sml:value></sml:Term></sml:identifier>
<sml:identifier name=\"shortName\"><sml:Term definition=\"urn:ogc:def:identifier:OGC:1.0:shortName\"><sml:value>52°North
GmbH</sml:value></sml:Term></sml:identifier></sml:IdentifierList></sml:identification><sml:capabilities name=\"offerings\">
<swe:SimpleDataRecord><swe:field name=\"Offering for sensor 9\"><swe:Text definition=\"urn:ogc:def:identifier:OGC:offeringID\">
<swe:value>http://www.52north.org/test/offering/9</swe:value></swe:Text></swe:field></swe:SimpleDataRecord></sml:capabilities>
<sml:capabilities name=\"parentProcedures\"><swe:SimpleDataRecord><swe:field name=\"parentProcedure\"><swe:Text>
<swe:value>http://www.52north.org/test/procedure/1</swe:value></swe:Text></swe:field></swe:SimpleDataRecord></sml:capabilities>
<sml:capabilities name=\"featuresOfInterest\"><swe:SimpleDataRecord><swe:field name=\"featureOfInterestID\"><swe:Text>
<swe:value>http://www.52north.org/test/featureOfInterest/9</swe:value></swe:Text></swe:field></swe:SimpleDataRecord></sml:capabilities>
<sml:position name=\"sensorPosition\"><swe:Position referenceFrame=\"urn:ogc:def:crs:EPSG::4326\"><swe:location><swe:Vector
gml:id=\"STATION_LOCATION\"><swe:coordinate name=\"easting\"><swe:Quantity axisID=\"x\"><swe:uom code=\"degree\"/>
<swe:value>7.651968812254194</swe:value></swe:Quantity></swe:coordinate><swe:coordinate name=\"northing\"><swe:Quantity axisID=\"y\">
<swe:uom code=\"degree\"/><swe:value>51.935101100104916</swe:value></swe:Quantity></swe:coordinate><swe:coordinate name=\"altitude\">
<swe:Quantity axisID=\"z\"><swe:uom code=\"m\"/><swe:value>52.0</swe:value></swe:Quantity></swe:coordinate></swe:Vector></swe:location>
</swe:Position></sml:position><sml:inputs><sml:InputList><sml:input name=\"test_observable_property_9\"><swe:ObservableProperty
definition=\"http://www.52north.org/test/observableProperty/9\"/></sml:input></sml:InputList></sml:inputs><sml:outputs><sml:OutputList>
<sml:output name=\"test_observable_property_9_1\"><swe:Category definition=\"http://www.52north.org/test/observableProperty/9_1\">
<swe:codeSpace xlink:href=\"NOT_DEFINED\"/></swe:Category></sml:output><sml:output name=\"test_observable_property_9_2\"><swe:Count
definition=\"http://www.52north.org/test/observableProperty/9_2\"/></sml:output><sml:output name=\"test_observable_property_9_3\">
<swe:Quantity definition=\"http://www.52north.org/test/observableProperty/9_3\"><swe:uom code=\"NOT_DEFINED\"/></swe:Quantity>
</sml:output><sml:output name=\"test_observable_property_9_4\"><swe:Text
definition=\"http://www.52north.org/test/observableProperty/9_4\"/></sml:output><sml:output name=\"test_observable_property_9_5\">
<swe:Boolean definition=\"http://www.52north.org/test/observableProperty/9_5\"/></sml:output></sml:OutputList></sml:outputs>
</sml:System></sml:member></sml:SensorML>",
  "observableProperty": [
    "http://www.52north.org/test/observableProperty/9_1",
    "http://www.52north.org/test/observableProperty/9_2",
    "http://www.52north.org/test/observableProperty/9_3",
    "http://www.52north.org/test/observableProperty/9_4",
    "http://www.52north.org/test/observableProperty/9_5",
    "http://www.52north.org/test/observableProperty/9_6",
    "http://www.52north.org/test/observableProperty/9_9"
  ],
  "observationType": [
    "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_Measurement",
    "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_CategoryObservation",
    "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_CountObservation",
    "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_TextObservation",
    "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_TruthObservation",
    "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_GeometryObservation",
    "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_ReferenceObservation"
  ],
  "featureOfInterestType": "http://www.opengis.net/def/samplingFeatureType/OGC-OM/2.0/SF_SamplingPoint"
}
```


Figure 85: 52 North SOS request for "InsertSensor" operation

Appendix B - 2. 52° North Sensor Observation Service "InsertResultTemplate" operation request


```
{
  "request": "InsertResultTemplate",
  "service": "SOS",
  "version": "2.0.0",
  "identifier": "http://www.52north.org/test/procedure/9/template/1",
  "offering": "http://www.52north.org/test/offering/9",
  "observationTemplate": {
    "type": "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_Measurement",
    "procedure": "http://www.52north.org/test/procedure/9",
    "observedProperty": "http://www.52north.org/test/observableProperty/9_3",
    "featureOfInterest": {
      "identifier": {
        "value": "http://www.52north.org/test/featureOfInterest/9",
        "codespace": "http://www.opengis.net/def/nil/OGC/0/unknown"
      },
      "name": [
        {
          "value": "52°North",
          "codespace": "http://www.opengis.net/def/nil/OGC/0/unknown"
        }
      ],
      "sampledFeature": [
        "http://www.52north.org/test/featureOfInterest/world"
      ],
      "geometry": {
        "type": "Point",
        "coordinates": [
          51.935101100104916,
          7.651968812254194
        ],
        "crs": {
          "type": "name",
          "properties": {
            "name": "EPSG:4326"
          }
        }
      }
    },
    "phenomenonTime": "template",
    "resultTime": "template",
    "result": ""
  },
  "resultStructure": {
    "fields": [
      {
        "type": "time",
        "name": "phenomenonTime",
        "definition": "http://www.opengis.net/def/property/OGC/0/PhenomenonTime",
        "uom": "http://www.opengis.net/def/uom/ISO-8601/0/Gregorian"
      },
      {
        "type": "quantity",
        "name": "observable_property_9",
        "definition": "http://www.52north.org/test/observableProperty/9_3",
        "uom": "test_unit_9"
      }
    ]
  },
  "resultEncoding": {
    "tokenSeparator": ",",
    "blockSeparator": "#"
  }
}
```


Figure 86: 52 North SOS request for "InsertResultTemplate" operation

Appendix B - 3. 52° North Sensor Observation Service "InsertResult" operation request

Request

POST | application/json | application/json | Permalink | Syntax

```
{
  "request": "InsertResult",
  "service": "SOS",
  "version": "2.0.0",
  "templateIdentifier": "http://www.52north.org/test/procedure/9/template/1",
  "resultValues": "15#2012-11-19T13:30:00+01:00,159.15#2012-11-19T13:31:00+01:00,159.15#2012-11-19T13:32:00+01:00,159.85#2012-11-19T13:33:00+01:00,160.5#2012-11-19T13:34:00+01:00,160.9#2012-11-19T13:35:00+01:00,160.7#2012-11-19T13:36:00+01:00,160.5#2012-11-19T13:37:00+01:00,160.6#2012-11-19T13:38:00+01:00,160.5#2012-11-19T13:39:00+01:00,160.4#2012-11-19T13:40:00+01:00,160.34#2012-11-19T13:41:00+01:00,160.25#2012-11-19T13:42:00+01:00,159.79#2012-11-19T13:43:00+01:00,159.56#2012-11-19T13:44:00+01:00,159.25#"
}
```

Figure 87: 52 North SOS request for "InsertResult" operation

Appendix B - 4. The initial values of the "Sensors" entity in JSON file for importing to SensorThings API server

```
[
  {
    "name": "eBike1-TCU",
    "description": "Telematic & Connectivity Unit sending ebike data",
    "encodingType": "application/pdf",
    "metadata": "https://www.smart.com/content/dam/smart/EN/PDF/smart_ebikeFlyer_2013_E-int.pdf"
  },
  {
    "name": "eBike2-TCU",
    "description": "Telematic & Connectivity Unit sending ebike data",
    "encodingType": "application/pdf",
    "metadata": "https://www.smart.com/content/dam/smart/EN/PDF/smart_ebikeFlyer_2013_E-int.pdf"
  },
  {
    "name": "eBike0C-TCU",
    "description": "Telematic & Connectivity Unit sending ebike data",
    "encodingType": "application/pdf",
    "metadata": "https://www.smart.com/content/dam/smart/EN/PDF/smart_ebikeFlyer_2013_E-int.pdf"
  },
  {
    "name": "eBike0D-TCU",
    "description": "Telematic & Connectivity Unit sending ebike data",
    "encodingType": "application/pdf",
    "metadata": "https://www.smart.com/content/dam/smart/EN/PDF/smart_ebikeFlyer_2013_E-int.pdf"
  },
  {
    "name": "Garmin Smart Watch fenix 5X",
    "description": "Garmin Multisport GPS Watch for measure health of ebike user in i_city project",
    "encodingType": "application/pdf",
    "metadata": "http://static.garmin.com/pumac/fenix5x_OM_EN.pdf"
  }
]
```

Figure 88: The initial values of the "Sensors" entity in JSON file for importing to SensorThings API server

Appendix B - 5. The example initial values of the "Things" entity in JSON file for importing to SensorThings API server

```
[
{
    "name": "eBike1",
    "description": "Electric Bike #1 (i_city project)",
    "properties": {
        "vin": "eBike20131126003c"
    }
},
{
    "name": "eBike2",
    "description": "Electric Bike #2 (i_city project)",
    "properties": {
        "vin": "eBike201209280029"
    }
},
{
    "name": "eBike0C",
    "description": "Electric Bike #0C (i_city project)",
    "properties": {
        "vin": "eBike20131107003d"
    }
},
{
    "name": "eBike0D",
    "description": "Electric Bike #0D (i_city project)",
    "properties": {
        "vin": "eBike201311270017"
    }
},
{
    "name": "Garmin Smart Watch fenix 5X",
    "description": "Garmin Multisport GPS Watch for measure health of ebike user in i_city
   project",
    "properties": {
        "Brand": "Garmin",
        "UnitID" : "3952924420",
        "ProductID" : "2604",
        "Sensors" : " GPS/GLONASS/wrist heart rate monitor/Barometric
   altimeter/Compass/Gyroscope/Accelerometer/Thermometer"
    }
}
]
```

Figure 89: The initial values of the "Things" entity in JSON file for importing to SensorThings API server

Appendix B - 6. The example of the initial value of the "ObservedProperties" entity of the Smart Electric Bike in JSON file for importing to SensorThings API server

```
[{
  "name": "eb.fuelLevel",
  "description": "The incoming ebike data indicates the amount of remaining bicycle battery
   provided in % (De: Der von der Batterie bereitgestellte ladestand in %. Kann unter last
   schwanken!)",
  "definition": "https://www.hft-
   stuttgart.de/Forschung/Kompetenzen/wf/Projekte/Projekt195.html/en"
},
{
  "name": "eb.batteryVoltageBike",
  "description": "The incoming ebike data indicates voltage of the remaining bicycle
   battery in milli-volt. (De: Spannung der Fahrradbatterie in millivolt.!)",
  "definition": "https://www.hft-
   stuttgart.de/Forschung/Kompetenzen/wf/Projekte/Projekt195.html/en"
},
{
  "name": "eb.mileage",
  "description": "The incoming ebike data indicates driven distance of the bike since the
   TCU switched on (De: Zeigt an wie viel Meter das Fahrrad mit angeschalteter TCU
   gefahren ist)",
  "definition": "https://www.hft-
   stuttgart.de/Forschung/Kompetenzen/wf/Projekte/Projekt195.html/en"
},
{
  "name": "eb.pedalForce",
  "description": "The incoming ebike data indicates the pedal force [0 - ~40] (De: Zeigt an
   wie stark der Fahrer in die Pedale Tritt)",
  "definition": "https://www.hft-
   stuttgart.de/Forschung/Kompetenzen/wf/Projekte/Projekt195.html/en"
},
...
...
]
```

Figure 90 : The example of the first four initial values of the "ObservedProperties" entity of the Smart Electric Bike in JSON file for importing to SensorThings API server

Appendix B - 7. The example of the initial values of the "ObservedProperties" entity of the Garmin Smart Watch in JSON file for importing to SensorThings API server

```
[
    {
  "name": "gsw.TCX.distanceMeter",
  "description": "Altitude data",
  "definition": "http://static.garmin.com/pumac/fenix5x_OM_EN.pdf"
},
{
  "name": "gsw.TCX.HeartRateBpm",
  "description": "HeartRateBpm",
  "definition": "http://static.garmin.com/pumac/fenix5x_OM_EN.pdf"
},
{
  "name": "gsw.TCX.Speed",
  "description": "Speed",
  "definition": "http://static.garmin.com/pumac/fenix5x_OM_EN.pdf"
},
{
  "name": "gsw.GPX.Temp",
  "description": "Temperature",
  "definition": "http://static.garmin.com/pumac/fenix5x_OM_EN.pdf"
},
...
...
]
```

Figure 91: The example of the first four initial values of the "ObservedProperties" entity of the Garmin Smart Watch in JSON file for importing to SensorThings API server

Appendix B - 8. The initial value of the "Locations" entity in JSON file for importing to SensorThings API server

```
{
    "name": "HFT Stuttgart",
    "description": "UNIVERSITY OF APPLIED SCIENCES STUTTGART, GERMANY (Starting Point)",
    "encodingType": "application/vnd.geo+json",
    "location": {
      "type": "Point",
      "coordinates": [9.1726, 48.7803,250]
    }
  }
```

Figure 92: The initial value of the "Locations" entity in JSON file for importing to SensorThings API server (specifying the geospatial location in 3D with GeoJSON)

Appendix B - 9. Relation table of the "Datastreams" entity to "ObservedProperties", "Sensors" and "Things" entities in SensorThings API server

Table 15: The table of relation of the "Datastreams" entity to "ObservedProperties", "Sensors" and "Things" in SensorThings API server

Datastream ID	ObservedProperties	Sensors/Things ID	ID
1	fuelLevel	1	E-bike20131126003c
2	fuelLevel	2	E-bike201209280029
3	fuelLevel	3	E-bike20131107003d
4	fuelLevel	4	E-bike201311270017
5	batteryVoltagE-bike	1	E-bike20131126003c
6	batteryVoltagE-bike	2	E-bike201209280029
7	batteryVoltagE-bike	3	E-bike20131107003d
8	batteryVoltagE-bike	4	E-bike201311270017
9	mileage	1	E-bike20131126003c
10	mileage	2	E-bike201209280029
11	mileage	3	E-bike20131107003d
12	mileage	4	E-bike201311270017
13	pedalForce	1	E-bike20131126003c
14	pedalForce	2	E-bike201209280029
15	pedalForce	3	E-bike20131107003d
16	pedalForce	4	E-bike201311270017
17	speed	1	E-bike20131126003c
18	speed	2	E-bike201209280029
19	speed	3	E-bike20131107003d
20	speed	4	E-bike201311270017
21	light	1	E-bike20131126003c
22	light	2	E-bike201209280029
23	light	3	E-bike20131107003d
24	light	4	E-bike201311270017
25	charging	1	E-bike20131126003c
26	charging	2	E-bike201209280029
27	charging	3	E-bike20131107003d
28	charging	4	E-bike201311270017
29	motorSupportmin	1	E-bike20131126003c
30	motorSupportmin	2	E-bike201209280029
31	motorSupportmin	3	E-bike20131107003d
32	motorSupportmin	4	E-bike201311270017

Datastream ID	ObservedProperties	Sensors/Things ID	ID
33	motorSupport.max	1	E-bike20131126003c
34	motorSupport.max	2	E-bike201209280029
35	motorSupport.max	3	E-bike20131107003d
36	motorSupport.max	4	E-bike201311270017
37	motorSupport.avg	1	E-bike20131126003c
38	motorSupport.avg	2	E-bike201209280029
39	motorSupport.avg	3	E-bike20131107003d
40	motorSupport.avg	4	E-bike201311270017
41	motorSupport.firstVal	1	E-bike20131126003c
42	motorSupport.firstVal	2	E-bike201209280029
43	motorSupport.firstVal	3	E-bike20131107003d
44	motorSupport.firstVal	4	E-bike201311270017
45	motorSupport.firstValTime	1	E-bike20131126003c
46	motorSupport.firstValTime	2	E-bike201209280029
47	motorSupport.firstValTime	3	E-bike20131107003d
48	motorSupport.firstValTime	4	E-bike201311270017
49	motorSupport.lastVal	1	E-bike20131126003c
50	motorSupport.lastVal	2	E-bike201209280029
51	motorSupport.lastVal	3	E-bike20131107003d
52	motorSupport.lastVal	4	E-bike201311270017
53	motorSupport.count	1	E-bike20131126003c
54	motorSupport.count	2	E-bike201209280029
55	motorSupport.count	3	E-bike20131107003d
56	motorSupport.count	4	E-bike201311270017
57	motorSupport.timeSpan	1	E-bike20131126003c
58	motorSupport.timeSpan	2	E-bike201209280029
59	motorSupport.timeSpan	3	E-bike20131107003d
60	motorSupport.timeSpan	4	E-bike201311270017
61	uuid	1	E-bike20131126003c
62	uuid	2	E-bike201209280029
63	uuid	3	E-bike20131107003d
64	uuid	4	E-bike201311270017
65	geo.accuracy	1	E-bike20131126003c
66	geo.accuracy	2	E-bike201209280029
67	geo.accuracy	3	E-bike20131107003d
68	geo.accuracy	4	E-bike201311270017
69	geo.latitude	1	E-bike20131126003c

Datastream ID	ObservedProperties	Sensors/Things ID	ID
70	geo.latitude	2	E-bike201209280029
71	geo.latitude	3	E-bike20131107003d
72	geo.latitude	4	E-bike201311270017
73	geo.longitude	1	E-bike20131126003c
74	geo.longitude	2	E-bike201209280029
75	geo.longitude	3	E-bike20131107003d
76	geo.longitude	4	E-bike201311270017
77	geo.altitude	1	E-bike20131126003c
78	geo.altitude	2	E-bike201209280029
79	geo.altitude	3	E-bike20131107003d
80	geo.altitude	4	E-bike201311270017
81	Garmin-TCX: DistanceMeters	5	3952924420
82	Garmin-TCX: HeartRateBpm	5	3952924420
83	Garmin-TCX: Speed	5	3952924420
84	Garmin-GPX: Temperature	5	3952924420
85	Garmin-TCX: latitude	5	3952924420
86	Garmin-TCX: longitude	5	3952924420
87	Garmin-TCX: altitude	5	3952924420
88	OpenWeatherData: Temperature (Kelvin)	6	2825297
89	OpenWeatherData: Pressure	6	2825297
90	OpenWeatherData: humidity	6	2825297
91	OpenWeatherData: WindSpeed	6	2825297
92	OpenWeatherData: WindDegree	6	2825297
95	OpenWeatherData: Cloudiness	6	2825297
96	OpenWeatherData: Temperature (Celcius)	6	2825297

Appendix C. Appendix Disc

Table 16: Appendix Disc File Description

Disc directory	Description
Programming for Sensor Network	
./SensorNetwork/1_Post_Things.js	NodeJS program to insert new "Things" Entity to the SensorThings API server from JSON file containing many objects.
./SensorNetwork/2_Post_Sensors.js	NodeJS program to insert new "Sensors" Entity to the SensorThings API server from JSON file containing many objects.
./SensorNetwork/3_Post_ObsProp.js	NodeJS program to insert new "ObservedProperties" Entity to the SensorThings API server from JSON file containing many objects.
./SensorNetwork/4_Post_DS.js	NodeJS program to insert new "Datastreams" Entity to the SensorThings API server from JSON file containing many objects.
./SensorNetwork/5_Post_Locations.js	NodeJS program to insert new "Locations" Entity to the SensorThings API server from JSON file containing many objects.
./SensorNetwork/6_Post_FOI.js	NodeJS program to insert new "FeatureOfInterest" Entity to the SensorThings API server from JSON file containing many objects.
./SensorNetwork/7_Post_Obs.js	NodeJS program to insert new "Observations" Entity to the SensorThings API server from JSON file containing many objects.
./SensorNetwork/DeleteService.js	NodeJS program to delete the "Sensors", "Things", "Datastreams", "Locations", "ObservedProperties", "FeaturesOfInterest" and "Observations" Entities with a specified range of the entity id.
./SensorNetwork/Aggregation_ServerSide.js	NodeJS program to request the observation result from the sensor network and then do the aggregation on the server-side. (Only for test the server-side aggregation capability)
./SensorNetwork/CheckDatastreamID.js	NodeJS program to check the Datastream ID of the input Observed property name and Things ID.
./SensorNetwork/CZML_Builder.js	NodeJS program to build the Cesium file (CZML) on the server-side. (Only for test purpose)
Programming for Cesium Application	
./Project_Cesium/index.html	Main index HTML file of the i_city ebike sharing application
./Project_Cesium/js/icitymainapp.js	JavaScript programming of the main 3D web-based application based on Cesium JavaScript library
./Project_Cesium/js/icityMapPin.js	JavaScript programming to create new Cesium Map Pin entity with a specified Maki ID and pixel size from the JSON or table which contain the data name, Latitude, and Longitude.
./Project_Cesium/js/getSTA.js	JavaScript programming to get the data from Sensor network with specified start time, end time, IoT id or Datastream id

Disc directory	Description
./Project_Cesium/js/DrawChart.js	JavaScript programming for draw a Chart using Highchart JS library using the data from the Sensor network
./Project_Cesium/js/DrawStat.js	JavaScript programming for computing the parameter statistic using JStat JS library using the data from the Sensor network
./Project_Cesium/js/Real_Time_Ebike.js	JavaScript programming for updating the real-time status of each E-bike
Demo of i_city E-bike Sharing Application	
./ApplicationDemo/i_city_ebikeSharing.mp4	A video file of "i_city E-bike sharing application" demo
./ApplicationDemo/iCity_3D_glTF.png	A figure in png format showing "i_city E-bike sharing application" screenshot with a functionality to display 3D city model in glTF format
./ApplicationDemo/iCity_3DTile.png	A figure in png format showing "i_city E-bike sharing application" screenshot with a functionality to display 3D city model in Cesium 3D-Tile format
./ApplicationDemo/iCity_HistoricalStatistic.png	A figure in png format showing "i_city E-bike sharing application" screenshot with a functionality to display Historical Statistic of E-bike usage data
./ApplicationDemo/iCity_HistoricalTrack.png	A figure in png format showing "i_city E-bike sharing application" screenshot with a functionality to display Historical E-bike route in 3D environment
./ApplicationDemo/iCity_Mappin.png	A figure in png format showing "i_city E-bike sharing application" screenshot with a functionality to display various types of Map Pin
./ApplicationDemo/iCity_RealTime.png	A figure in png format showing "i_city E-bike sharing application" screenshot with a functionality to display the latest E-bike status