Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Petter Selfors Rølvåg

Structural Health Monitoring of Mountain Bike

Master's thesis in Engineering and ICT
Supervisor: Bjørn Haugen
Co-supervisor: Terje Rølvåg
January 2022

NTNU
Norwegian University of
Science and Technology

Petter Selfors Rølvåg

Structural Health Monitoring of Mountain Bike

Master's thesis in Engineering and ICT
Supervisor: Bjørn Haugen
Co-supervisor: Terje Rølvåg
January 2022

Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Abstract

An approach for the implementation of real time digital twin (DT) based structural health monitoring (SHM) of bicycle frames is presented in this thesis. The primary goal is to establish an Informational Technology (IT) system that can support SHM via the use of Internet of Things (IoT) technology, as well as to apply and develop DT methodologies. When developing the server and client solutions, the authors utilized Mendix, a platform-as-a-service solution, as the primary development tool. The authors used an Arduino to digitize the real loads, accelerations, and strain time histories that the bicycle is experiencing. The smart phone is connected to the Arduino through Bluetooth, and the data from the sensors is consumed before being uploaded to the cloud. Due to the fact that the DT approaches are cloud-based and consume sensor data, they help to develop an Internet of Things ecosystem with other devices such as the smart phone and Arduino, which all produce and consume data from the same database. The digital twin is represented by a 6x8 look-up table, which allows real-time strain calculations in 8 virtual strain gauges to be performed during a bicycle ride. The 6x8 look-up table is precomputed using a unit-load approach applied to a bicycle frame modelled in a Finite Element (FE) software. On the basis of a single IMU sensor, an analytical inverse method for the determination of dynamic bicycle loads is built. Each column in the ROM represents the stress distribution in 8 gauges caused by one unit load. The real gauge stresses are calculated by multiplying the 6x8 matrix with the 6 loads calculated by the inverse method. Noise reduction and singularity removal are performed on the sensor outputs. For model calibration and validation, additional strain gauges and temporary accelerometers were put to the bicycle. The current technique is generic but linear and hence only applicable to undamped bicycle frames. The findings imply that there is potential for improvement in both the IoT system and inverse method. The list of possible enhancements is extensive, and the future possibilities are intriguing.

Acronyms

AI Artificial Intelligence. 1, 29

API Application Programming Interface. 17, 29, 30

AWS Amazon Web Services. 16, 29

CSV Comma-Separated Values. 31

DAC Digital-to-Analog Converter. 21

DT Digital Twin. i, 1–3, 36, 54

ER Entity Relationship. 31

FE Finite Element. i, 20

FEA Finite Element Analysis. 18

FEM Finite Element Method. 1, 44

FMU Functional Mockup Unit. 18

HBM Hottinger Baldwin Messtechnik GmbH. 26

HTTP Hypertext Transfer Protocol. 4, 7, 29, 53

IaaS Infrastructure as a Service. 16

IIoT Industrial Internet of Things. 1, 4, 5, 29, 53

IMU Inertial Measurement Unit. 23, 24, 36, 40, 43–45, 51, 53

IoT Internet of Things. i, iii, 1–7, 29–31, 53, 54, 58

IT Informational Technology. i, 2, 15, 54

ML Machine Learning. 1, 29

MQTT Message Queuing Telemetry Transport. 4, 7, 8, 54

MVC Model View Controller. 15

PaaS Platform as a Service. 15, 16, 34, 51, 53

PnP Plug-and-Play. 49, 54

RESTful REpresentational State Transfer. 4, 7, 29, 53

ROM Reduced Order Model. iii, 18–20, 29, 37, 44, 52, 54

SHM Structural Health Monitoring. 1–4, 6, 18, 23, 54

TCP Transmission Control Protocol. 7, 29, 30

UI User Interface. 32

Contents

Chapter 1

Introduction

1.1 Background and Motivation

Different load cases with seemingly infinite data points influence structures in various industries worldwide. Predicting when to maintain these structures is beneficial concerning both safety and cost. Currently, the maintenance is based on time-consuming manual inspection procedures, and the processes do not account for what happens in real-time. Structural Health Monitoring (SHM) procedures in maintenance and monitoring applications can assist with safety and serviceability through the use of real-time data.

It is crucial to understand how to manage the data when using digital transformation strategies to create Internet of Things (IoT) / Industrial Internet of Things (IIoT) or Digital Twin (DT) systems to support business processes. The end goal of a digital transformation strategy is to create these solutions, which have the means to sew together a piece of the business, or the entire business to become more responsive and work smarter. If the data is not handled correctly, the resulting information will not be precise enough and become less attractive to use. This means that it is essential to know what data is needed, how the data is being stored and in which frequency, where and when the data is needed, and how the applications access, process, and display the data. Raw data is not always helpful information and must be processed to become beneficial information. Not only does the information need to be processed, but it is also essential to know at which layer of the system the processing happens. Additionally, processed data is also helpful in generating new information. Therefore, technology such as Artificial Intelligence (AI) and Machine Learning (ML) is suitable for predictive maintenance and analyzing trends over a more extended period. With AI and ML technologies, the applications become smarter due to their self-learning nature. Other ways to analyze structures are by using FEM software, but it is very challenging to use in real-time because of the required processing power and licensing costs; thus, it does not scale very well.

DT-based SHM is a critical tool for the shift from reactive to predictive maintenance in applications such as automobiles, cranes, bridges, aircraft, and offshore structures. However, even the most sophisticated IoT cloud systems are not intended to operate in real time at the requisite sample rate. Additionally, IoT suppliers do not enable quick, complementary edge solutions nor do they give real-time decision assistance for engineers. This may be a temporary issue, but we are attempting to design a hybrid solution that utilizes a combination of cloud (server) and edge (local) technologies.

This thesis provides in-depth knowledge of SHM procedures. It is tailored for those with interests in fields related to mechanical, electronics, Informational Technology (IT) and mechatronics. The article showcases one way of creating an IT system to support various data processing methods in close to real-time, with the use of one edge solution and one solution on the cloud. This way, the system can be used for vehicles, cranes, bridges, aerospace, and offshore structures as long as there is an internet connection.

1.2 Research Questions

The purpose of this assignment is to identify and quantify current state-of-the-art IoT/DT systems, as well as to develop new features that allow real-time structural health monitoring, failure prediction, and decision support. There are several accessible benchmark apps, but candidates must always have full access to all data. As such, this paper will answer the following four research questions (RQ):

1. Select locations and install sensors on the mountain bike for structural and performance monitoring based on a DT.
2. Implement an IoT / edge / application framework for collection, and processing of physical and virtual sensors for bicycle monitoring and benchmarking.
3. Implement an inverse method for bicycle load detection in the generic framework
4. Benchmark the Digital twin solution with respect to performance, fatigue prediction and other failure modes bicycle frames.

RQ1 is concerned with the physical asset, with the location of sensors on the bike that will provide adequate performance monitoring of the bike's structural health. This involves hardware selection, installation on the bike, choosing sensors and calibrating them. To make use of the sensor data, an IoT framework must be developed, which is what RQ2 will handle. Work during the spring established a foundation that will be reinforced more when we address RQ2. Additionally, one of the primary challenges for RQ2 is enhancing the system's stability and performance in order to obtain a sufficient sample rate. In RQ3, the sensors will be employed in the inverse method for bicycle load detection, using algorithms considered as part of RQ2's IoT framework. Finally, RQ4 is focused with testing and demonstrating the effectiveness of the whole system as well as the DT methods.

1.3 Thesis Structure

Chapter 2 will discuss the techniques and results of a literature review conducted to explain critical concepts and terminology. The findings in Chapter 2 serve as the foundation for the remainder of the research. Chapter 3 was written as part of the spring project assignment and discusses thoughts on how to represent the DT and how software is produced. The SHM system and its components are introduced in Chapter 4, with the purpose of presenting an overview of a general system for structural health monitoring. Chapter 5 will detail the calibration and configuration of the various sensors. Chapter 6 presents the results of the experiments conducted to evaluate the DT approaches and the IoT system's operation. Chapter 7 is a discussion in which the whole system is evaluated for correctness, performance, and modifiability, as well as some areas for improvement. Chapter 8 concludes the study and briefly discusses critical topics for further research.

Chapter 2 will detail the methods and findings of a literature review conducted to clarify key concepts and terminology. Chapter 2's results provide the groundwork for the rest of the research. Chapter 3 discusses the ideas that are employed to represent the digital twin, as well as the development process. Chapter 4 introduces the SHM system and its components with the goal of providing an overview of a generic system for structural health monitoring. Chapter 5 discusses how to calibrate and configure the different sensors. Chapter 6 summarizes the results of tests undertaken to assess the DT methods and the performance of the IoT system. Chapter 7 provides a study of the whole system's accuracy, performance, and modifiability, as well as possible opportunities for development. Chapter 8 summarizes the study's findings and suggests relevant areas for future research.

Chapter 2

Literature Review

To provide an additional context for the choices made throughout the creation of the SHM system, a comprehensive review of the literature was conducted. This chapter will discuss the process of collecting and analyzing publications, as well as the findings of the literature review.

2.1 Collection and analysis of articles

This section will outline the process of literacy acquisition and literature evaluation. A comprehensive review of the literature was conducted for this objective. Systematic reviews of the literature vary from typical reviews in that they use a reproducible, scientific, and transparent search strategy [1, 2, 3]. By following the phases of the systematic mapping process, the literature review is completed and enhanced [2, 3].

The modified systematic mapping of the literature is separated into five distinct processes, each of which leads to a distinct outcome: (1) Research questions are defined, (2) Structured search is conducted, (3) All articles are screened, (4) Semi-structured search and new keywords is conducted using the abstracts, and (5) Data extraction and mapping is conducted. [3]

To supplement the basis of articles previously employed, scientific databases were consulted to compile all relevant papers pertaining to their search subject. The original search strings included the terms IoT Framework, IoT Platform, IoT Protocol, IIoT, Digital Transformation, REpresentational State Transfer (RESTful) HTTP, MQTT, IoT Ecosystem, and Industry 4.0.

As part of the article screening process, the search was reduced down by removing items that had little relation to the thesis' RQs. Included articles advanced to the next round. Abstracts of passed-through publications were evaluated in this step and were again rejected if they were not of high relevance to the study. The next stage, the semi-structured search, included the development of new keywords based on the articles provided and the articles discovered through the structured search in order to identify secondary sources from the primary sources [3]. While extracting data in the last stage, all basic information about each article was gathered using a spreadsheet, including the title of the piece, the author(s), the year, and the journal in which it was published. This methodical data mapping is critical for developing an overview of the articles and subjects examined. It aided the author's efforts in arranging the articles according to their references while also including comments and ratings.

2.2 Findings of the Literature Review

2.2.1 IoT and IIoT differences

IoT and IIoT are two buzzwords that refer to the concept of intelligent physical objects that are linked to one another over the internet and capable of communicating seemlessly [22]. Through the integration of machine sensors, middleware, software, and backend cloud computation and storage systems, the Industrial Internet enables improved visibility and insight into a business's operations and assets. As a consequence, it enables the transformation of corporate operational processes via the use of the findings obtained through sophisticated analytics interrogation of big data sets[8]. The industrial internet of things is defined as the ecosystem generated when businesses alter their operations digitally via the use of Industry 4.0 technologies [10].

The primary distinction between IoT and IIoT is in their intended application, with IoT technology often aimed at consumers, while IIoT technology is aimed at industrial users such as manufacturers and supply chains. Accuracy and precision for IIoT applications is higher than in IoT applications because industries need to have higher fault tolerant systems because they deal with giant machines. IIoT systems work in spaces such as aerospace, healthcare, etc. where the room for error is very low so the risk impact is very high in comparison to consumer-based IoT applications. The respective focus of IoT and IIoT demand specifications, determines the ground for two parameters: accuracy/precision and risk impact. The accuracy and precision of industrial grade applications should be higher because they sometimes deal with hazardous processes and impact many lives on the factory shop floor. An error could cause a company millions or even billions of dollars of losses. [17]

2.2.2 IoT Framework

IoT frameworks seek to simplify their internal networks by concealing the bulk of the underlying complexity and exposing data, interfaces, and functions that enable interoperability. Frameworks take away the networking complexity using a higher-level message transmission abstraction such as REST or publish-subscribe. [16]

The current difficulty with the IoT framework is a lack of standardization, despite the fact that numerous international bodies are striving to standardize it. The biggest hurdle to standardizing the IoT framework at the moment is that the IoT demands a different approach than a traditional system, and these challenges are now emphasized. The majority of standards either focus on broad themes or on their specific subjects. However, it is projected that the likelihood of complete compliance in the sector for some time, even after the standards are implemented, would be minimal. [16, Chapter 3]

IoT architectures in such frameworks have four key design objectives[16, Chapter 3]:

1. Reduce manufacturing time and accelerate the commercialization of IoT products.
2. Reduce the perceived complexity associated with IoT setup and operation.
3. Enhance application portability and interoperability.
4. Enhance modifiability, dependability, and maintainability.

2.2.3 IoT Platform

An IoT platform is a multi-layer technology that allows quick development of IoT applications by offering a set of pre-configured functions. It is an essential component of the Internet of Things since it enables communication between objects. Platforms provide a number of services, including the management of several hardware and software communication protocols, the security and authentication of devices and users, and the collection, presentation, and analysis of sensor data. Thus, an IoT platform must perform two functions. [20]

1. It is a platform for solving problems.
2. it is a platform that connects into an ecosystem of other technologies.

2.2.4 The IoT Ecosystem

When it comes to the IoT ecosystem, it is composed of so-called producers and consumers. The producer generates data, such as sensor data, and the consumer consumes it in order to generate additional information or provide context for the data to be used in real-time decision-making systems [10]. Smart devices, the first of the IoT's fundamental components, gather data from their surroundings, communicate with other devices through wired or wireless network technologies, and allow internet-based communication [12].

The primary components are devices, communication protocols, and server- and cloud-based structures for storing the acquired data [9]. It is critical to process, store, and analyze the data stored in the cloud infrastructure. The data may be fed into SHM machine learning algorithms and included into the structural analysis model, hence increasing the accuracy of the model-based approach. These technologies act as both consumers and producers of data, first consuming, then processing, and producing new information [19]. The choice of protocol is critical and challenging for manufacturers and customers, even while there are several protocols based on the same principles, each with its own characteristics, advantages and disadvantages, and not all are appropriate to all IoT applications[12].

2.2.5 IoT Protocols

IoT Network Protocols

A network protocol is a collection of communication rules and processes that all stations that exchange data across a network must follow. There are several network protocols, however they do not all serve the same purpose or operate in the same manner. WiFi and Bluetooth both have a range of up to 100 meters. WiFi uses WPA and WPA2, while Bluetooth may employ a shared secret through unique identifiers. Both protocols are market-ready, but for very different uses. WiFi is intended for use with small to medium-sized networks and with any device that has cellular connection. Bluetooth is intended for use in smaller networks, such as those used to send audio to headsets. [12]

Application Protocols

One of the two most widely used communication protocols are MQTT, and RESTful HTTP [15].

RESTful HTTP
HTTP is the most widely used message protocol on the web. It was created by the Internet Engineering Task Force and the World Wide Web Consortium and was adopted as a standard in 1997 [6]. HTTP is the foundation for data interchange in the RESTful Web architecture, which is built on the request/response paradigm [18]. Unlike MQTT, which utilizes topics to identify data communication between the client and the server, HTTP employs a Universal Resource Identifier (URI) to identify data communication between the client and the server [18]. HTTP is a text-based protocol that imposes no size constraints on the header portion or the message payload. The data sharing process begins with a semi-permanent session and supports both persistent and non-persistent connections. HTTP is a TCP protocol that utilizes TLS/SSL for security and does not support QoS (Quality of Service) [5]. HTTP is not primarily used for the IoT sector since it is a network resource-intensive protocol.

MQTT
At its core, MQTT is a machine-to-machine, lightweight communication protocol. Every message contains a topic, organized in a tree-like structure, to which the clients subscribe

or publish. The protocol was standardized by ISO/IEC 20922 and was further accepted as part of OASIS. The MQTT protocol was designed for asynchronous communication. The protocol's open-source application, called Mosquitto, is able to provide most of the standard features of the protocol. When compared to other protocols like HTTP, the MQTT protocol has a considerably smaller footprint, making MQTT, as stated above, much more suitable for resource-constrained environments. MQTT deployment are very difficult with increases of devices, but is also, as mentioned, one of the best protocols for communication.[21]

The LAMA and GIAN use cases demonstrate how the MQTT protocol may be used to solve problems. LAMA (Location Aware Messaging for Accessibility) is a technology that enables essential information to be shared between individuals and regions. GAIAN Database is a distributed federated database developed in Java that minimizes maintenance via the use of biologically inspired self-organization concepts. MQTT was chosen by the corporation because to its tailored architecture for applications such as transmitting telemetry data to and from space probes, which needs less bandwidth and battery power.[21]

A MQTT client is any device (ranging from a microcontroller to a full-fledged server) that runs a MQTT library and communicates to a MQTT broker across a network. The MQTT client is a device that connects to a broker via a network over a wireless network. Multiple clients may receive a single broker's message (one to many capabilities). This enables data sharing as well as device management and control. The broker acts like a post office, sending messages simultaneously to a large number of customers (one to many). The key advantages of using a MQTT broker are as follows: (1) Client connections are no longer at risk of being compromised or unsecure. (2) Scalable from a single to hundreds of systems. (3) All client connection statuses are handled and monitored, including security credentials and certificates. (4) Decreased pressure on the network without jeopardizing network security (cellular or satellite network)[21]

Chapter 3

Theory

The theories and procedures that will be used to address the research questions given in this master's thesis encompass a variety of technical fields, including structural analysis, software development, and mechatronics. An overview of these ideas will be provided. Moreover, a set of new concepts are added to create a better context around the developed system during this thesis in order to contextualize and corroborate the results.

In the first two sections, common bicycle failure modes and the DT methodologies, FEM, ROM and inverse method will be explained. All of the information in these sections is linked to RQ1, RQ3, and the benchmarking in RQ4. Second, three sections devoted to RQ3 describe the Mendix low code environment, as well as important considerations in software architecture, and the Bluetooth protocol.

3.1 Common bicycle failure modes to be detected by the SHM framework

The primary goal of this research is to give a framework for determining frame loads and stresses. Bicycle frames, according to the manufacturer, seldom break because they are normally conservatively dimensioned. Although most cyclists seek strong and dependable frames, the weight penalty is disliked by energetic and professional riders. As a result, Hardrocx is looking for optimal designs that provide optimum frame integrity while weighing the least amount of weight. The chosen bike (physical asset) is a Hardrocx Super Motard M4 in size 19" 3.1. A 3D CAD model was also given by the manufacturer, allowing for an exact digital twin replica. The bike features an undamped rigid aluminum frame that may be represented and solved using a linear FE model and solver.

Cracks in frame joints caused by manufacturing flaws are the most prevalent failure mechanisms [23]. Fractures may also develop in the center of a pipe as a result of undersized tubes used to save weight. However, overturning or collision creating strains over the yield

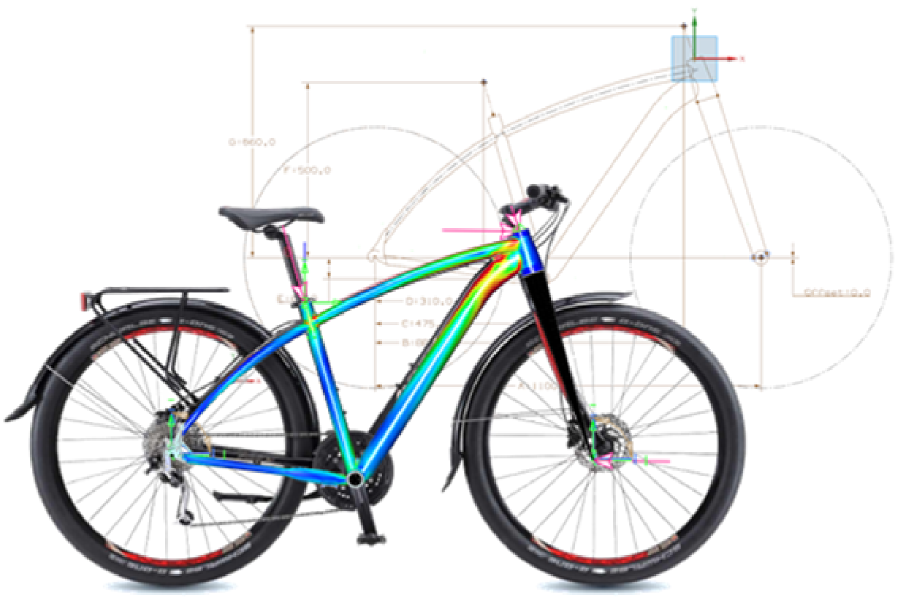


Fig. 3.1. The Hardrocx Super Motard M4.

limit is the most typical cause of frame breaking on an aluminum frame.

Seat tube fractures caused by consumers positioning seat pins too high are another example. If the seat pin does not extend far enough into the seat tube on the frame, the frame may break due to back and forth flexing. By monitoring the applied frame loads used in digital twin-based estimations of stress time histories and cumulative damage, the risk associated with these failure types may be mitigated.

3.2 Digital Twin Theory

To precompute the Reduced Order Model (ROM) look-up table, unit-loads were applied to the FE frame model in FEDEM (matrix). In real time, the ROM may be multiplied by the predicted load vector to determine gauge stresses and strains. FEDEM is also included in the cloud solution, allowing for further in-depth off-line stress measurement of the frame. Because of the continuing software connection between SAP (FEDEM) and SIEMENS, cloud calculations will be automated using Mendix. Despite the fact that these FEDEM formulations provide real-time gage stress calculations, a static ROM is employed since Mendix does not support real-time co-simulation. FEDEM is a multidisciplinary simulation system that enables integrated digital twin modeling and simulation by using a non-linear finite element formulation, CMS model reduction, and control system simulati$\Delta \boldsymbol{r}_k$ for time increment $\boldsymbol{k}$. In the non-linear situation, Newton-Raphson iterations must be employed to decrease the residual forces to attain equilibrium at the end of the time increment:

$$\boldsymbol{M}_k \Delta \ddot{\boldsymbol{r}}_k + \boldsymbol{C}_k \Delta \dot{\boldsymbol{r}}_k + \boldsymbol{K}_k \Delta \boldsymbol{r}_k = \Delta \boldsymbol{Q}_k \tag{3.1}$$

$\boldsymbol{M}_k$, $\boldsymbol{C}_k$, and $\boldsymbol{K}_k$ are the system mass, damping, and stiffness matrices at the start of time increment k, respectively. The system mass and stiffness matrices $\boldsymbol{M}_k$ and $\boldsymbol{K}_k$ are decreased using Component Mode Synthesis (CMS), which is the primary enabler for real-time FE modeling of non-linear systems such as entire mountain bikes [14].

$$\mathbf{v}_{free} = \begin{bmatrix} \mathbf{v}_e \\ \mathbf{v}_i \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{O} \\ \mathbf{B} & \mathbf{\Phi} \end{bmatrix} \begin{bmatrix} \mathbf{v}_e \\ \mathbf{y} \end{bmatrix} = \mathbf{H}\mathbf{v}_{\mathbf{sup}} \tag{3.2}$$

As stated in [13, 11], the same approach is applied to derive strain / stress time histories at chosen hotspots based on super node displacements.

$$\epsilon_{rosette} = [\boldsymbol{T_{re}}\ \widetilde{\boldsymbol{B}}\ \boldsymbol{T}\ \boldsymbol{A}\ \boldsymbol{L}\ \boldsymbol{H}]\boldsymbol{v_{sup}} \tag{3.3}$$

$\boldsymbol{H}$ is the CMS matrix that maps exterior displacements $\mathbf{v}_{\mathbf{sup}}$ to interior displacements $\mathbf{v}_{free}$. Internal displacements from linear couplings (MPCs) are recovered using the $\boldsymbol{L}$ matrix. The $\boldsymbol{A}$ matrix takes the complete displacement vector and extracts the nodal displacements that define the strain gauge. $\boldsymbol{T}$ is converting the extracted nodal displacements to strain gauge directions at the local level. The strain-displacement matrix provided by the derivatives of the strain element shape functions is the $\widetilde{\mathbf{B}}$ matrix. The $\mathbf{T}_{\mathbf{re}}$ matrix, which is optional, translates the estimated rosette strains and stresses to user-specified directions.

Each strain gauge element's $[\boldsymbol{T_{re}}\ \widetilde{\boldsymbol{B}}\ \boldsymbol{T}\ \boldsymbol{A}\ \boldsymbol{L}\ \boldsymbol{H}]$ may be precomputed, allowing for quick real-time computations of hot spot stresses during crane operations. As a result, this formulation is relevant to digital twin / hardware in the loop applications.

3.3 Software Development

During software development, there are benefits from picking suitable quality attributes to focus on critical aspects of the system [4]. Regarding this system, there are two quality attributes which are particularly interesting; performance and modifiability. [23]

3.3.1 Software Lifecycle

When developing complex software, having a well-considered software architecture[4] is critical. A software architecture is composed of architectural- or design patterns that are utilized to achieve business goals. Technology and project scope has to be evaluated as well. Architectural patterns define how software will take shape and may evolve over time. Enriching data and contextualizing it is a critical connection between machines and companies, and the design must enable this as well to build a smart system. [23]

Selecting relevant patterns and thinking like a software developer is therefore critical for supporting the company and making it possible to do more with less. Having architectural documentation helps a developer to decide where extra functionality inside the framework may be added, hence saving time and money. The ultimate aim is to minimize implementation time, to make modifications more affordable, to build more adaptable systems, to make scaling more affordable, and to avoid delays in order to support business choices both now and in the future. [23]

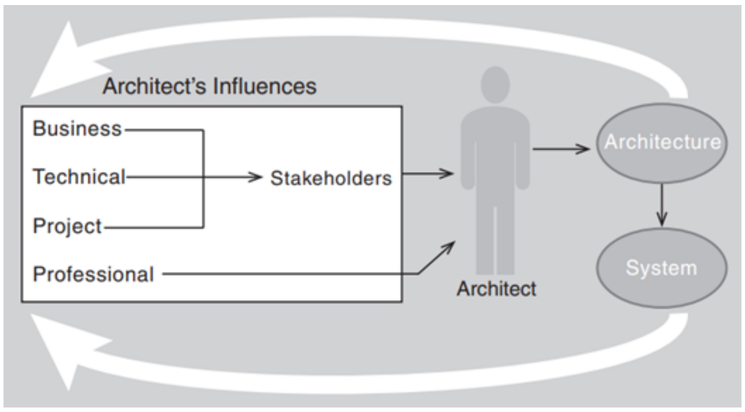


Fig. 3.2. Arrows denote the architecture's flow. Each cycle begins with influences that are weighed by the architect who designs the building. The system emerges from the architecture, which in turn generates new influences. [4]

3.3.2 Performance

In the software domain, performance is described as an event arriving at the system[4]. The event creates a response, thus, consumes time and resources. Throughout all of this, the system may also service other events. A modern system for structural health monitoring needs sufficiently low latency and a high throughput.[4]

Performance tactics, which can be seen in Figure 3.3, may be implemented to ensure that the system meets time requirements. Meeting timing requirements demands a response to an event arriving in the system in a specific time span [4]. An example is when a bicyclist starts logging their trip and see the logged data in a graph. This scenario includes several events such as listening for changes in a Bluetooth characteristic value, or retrieving a response from the server that an object has been created. There may also be specific events only running on the server, for instance running a piece of java code every time some data is committed to the server. When too many events occur at the same time, the experience may be reduced drastically for the end user. [4]

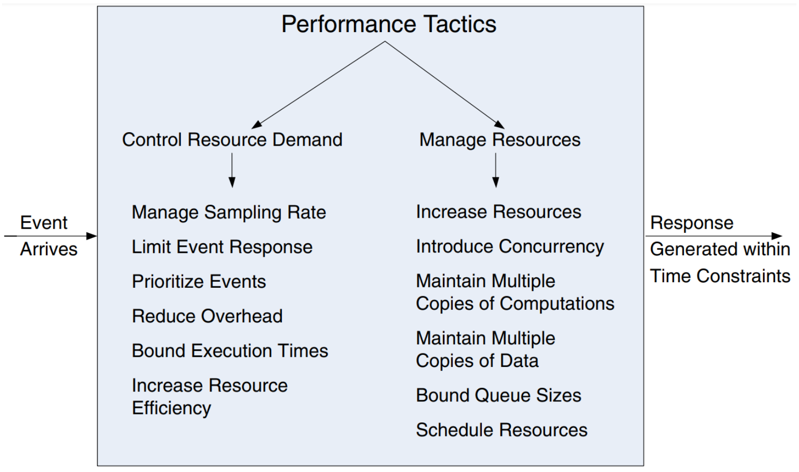


Fig. 3.3. A variety of tactics that may be used to enhance the system's performance. [4]

Regarding system latency, latency is the time between event arrival and the granting of the response. An event is either processed or blocked when it arrives at the system, resulting in two main factors contributing to system latency - resource consumption and blocked time. Resource consumption is the time the system takes to process the given information. Blocked time is the time needed to re-send the data and then process it. For maximizing performance, there are tactics called resource demand and resource management[4]

Resource demand is the frequency of events and how much resources each event consumes. Implementing- or optimizing algorithms to minimize the resources required to process each event will therefore reduce latency. However, this implies removing layers of code and a decrease in modifiability, which is a common trade-off when optimizing[4].

It is possible to manage resources by increasing the processing time of an event or by managing the resource and then sending it to the server. A reduction in the system latency follows with the use of these tactics. Additionally, different ways to manage resources are to increase available resources, cache data, distribute computations, or use concurrency [4].

3.3.3 Modifiability

Many studies shows that costs in typical software systems occurs after initial release[4]. Changes in software are made to fix bugs, improve performance or enhance user experience. The interest in having a modifiable software centers around the cost and risk of making changes, thus, it is important to consider the four questions from page 117 in [4].

- What can change?
- What is the likelyhood of the change?
- When is the change made and who makes it?
- What is the cost of the change?

Tactics concerning modifiability, as shown in Figure 3.4, aim to control the complexity of making changes and the time and cost it would consume. The two main ideas for achieving modifiability are increasing cohesion and decreasing coupling. This can be made by asserting the binding time of modification, lowering the coupling between the different modules of the system, increasing cohesion inside each module, and having well-documented code. [4]

A notable tactic regarding this thesis is preparing software for late binding. Late binding reduces costs by a large margin but requires an upfront cost due to an increase in development time. An example of taking advantage of late binding is setting the unique Bluetooth characteristic identifier of a Bluetooth device in run time. In this case, binding in run time reduces the high coupling caused by the Bluetooth architecture. However, this also implies that the system must support reading any data type stored in a Bluetooth characteristic of any device - which is an almost impossible task. Thus, the supported data type must be pre-programmed by the system developer. [4]

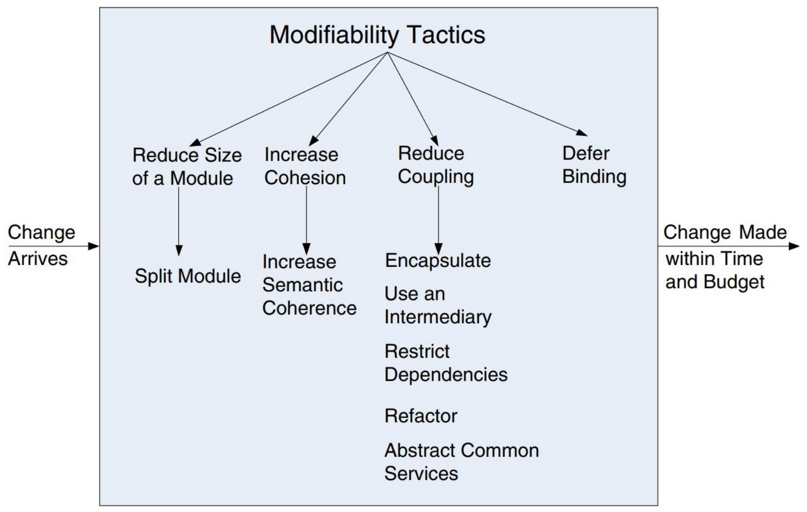


Fig. 3.4. Modifiability tactics improves the system modifiability. [4]

3.4 Mendix: Low-code Application Development Platform

Companies have invested in platforms to streamline development time. Mendix, which is a PaaS is one of these services that allows faster software creation by abstracting and automating parts of the development process. These services are ideal for organizations who seek agility within IT. Many applications existing today follow the same pattern called Model View Controller (MVC), which is very popular among todays websites and applications[4]. The MVC pattern is the essence of the Mendix platform and allows the developer to access solutions that support high modifiability and scalability. This way, developers avoid the need to reinvent the wheel since PaaS reduces the costs and complexity of managing infrastructure so that users can focus on improving the application delivery process. MVC increases decoupling between the different parts of the system such that they can be developed and tested individually without affecting one another. One can observe this pattern in Mendix through its components called action, flow, and page. Interoperating with other system is possible through the java or javascript actions. Mendix enables users to model an application in a human-readable form, reducing the complexity and turnaround time for custom development. Learning tools on their websites allow Mendix developers to learn the system through different modules [34].

Figure 3.5 depicts the Mendix Studio Pro 9.5.0 desktop application for designing client and server solutions. This tool enables full-stack software development by using pre-configured components for page and database development. Additionally, the program offers an effective error handling mechanism that presents potential error fixes. Additionally, there is a marketplace where developers may share and install modules.

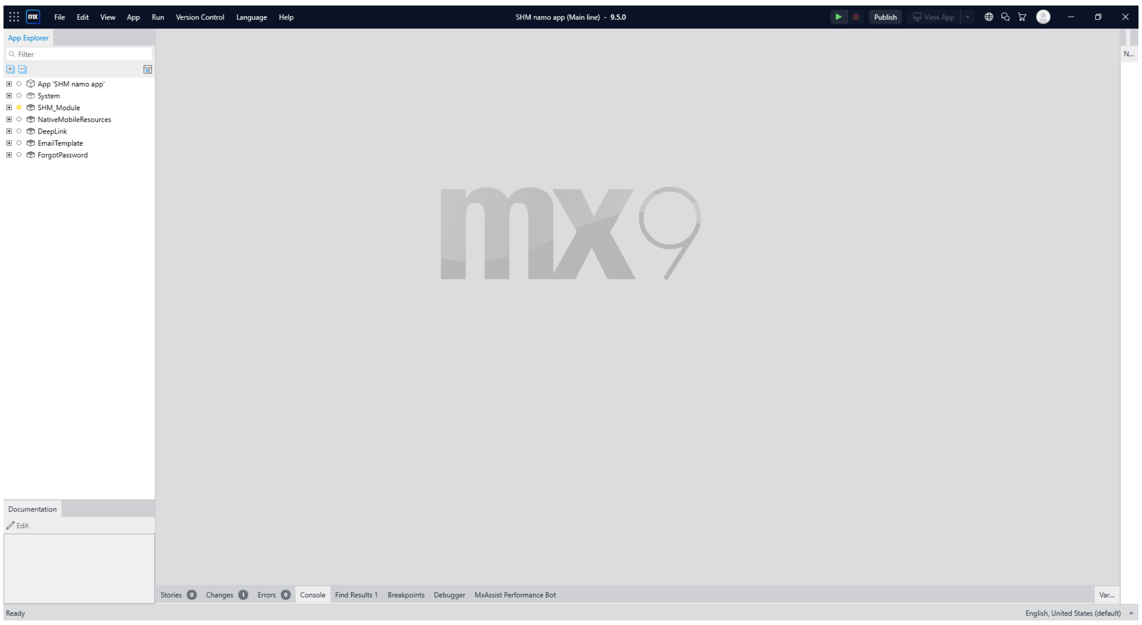


Fig. 3.5. The Mendix Studio Pro 9.5.0 PaaS desktop application for developing client and server solutions.

3.4.1 Cloud Foundry

Cloud Foundry is an open source cloud Platform as a Service (PaaS) designed specifically for developers. Cloud Foundry, in contrast to other cloud platform services, is a self-contained software package. Cloud Foundry is available on AWS, Azure, and Google Cloud, as well as on-premises through OpenStack, HP's Helion, or VMware's vSphere. Cloud Foundry includes routing, authentication, application lifecycle management, storage and execution, service brokers, messaging, monitoring, and logging. [28]

Cloud Foundry can help developers manage their workloads and Linux application resources more efficiently, hence saving operating expenses [28]. It supports DevOps workflows and multi-tenant computing. Cloud Foundry manages the lifetime of applications, including their development, testing, and deployment, as well as their interface with cloud providers. It is a free Platform as a Service (PaaS) that enables developers to create in a variety of languages. This helps to avoid vendor lock-in. Cloud Foundry supports the following programming languages: Java, Node.js, Go, PHP, Python, Ruby, .NET Core, and Staticfile. VMware invented Cloud Foundry, which is now owned by Dell Technologies' Pivotal Software. The Cloud Foundry Foundation is a non-profit collaborative project of the Linux Foundation and backed by companies such as IBM, SUSE, SAP, VMWARE, Accenture, Huawei and more [29].

3.4.2 Amazon Web Services

When it comes to cloud ecosystems, Amazon Web Services (AWS) is the market leader in both Infrastructure as a Service (IaaS) and Platform as a Service (PaaS), which can be combined to create scalable cloud applications without having to worry about infrastructure provisioning (compute, storage, and network) or management delays. [24]

Through the usage of AWS, one may choose the precise solutions that are required and pay only for the infrastructure that is really utilized, resulting in cheaper capital investment and quicker time to value without compromising application performance or user experience. One of the reasons why many organizations utilize AWS is that it provides a variety of storage options that are both affordable and quickly accessible. It may be used for a variety of tasks like as data storage and file indexing, as well as to operate mission-critical business applications. [24]

API-driven programming on AWS may allow businesses to construct uncompromisingly scalable apps without the need for an operating system or other technologies. There are so many firms all over the globe that use AWS to build, implement, and host applications, whether they are technological giants, startups, the government, food producers, or retail enterprises. In accordance with Amazon, the number of active AWS users has surpassed one million. Netflix, Adobe, Airbnb, AOL, and Coinbase are just a few of the firms that rely on AWS. [24]

3.4.3 Application Programming Interface

An Application Programming Interface (API) allows businesses to expose the data and functionality of their applications to external third-party developers, commercial partners, and internal departments inside their organizations. Through a specified interface, this enables services and products to interact with one another and exploit one other's data and capability. Developers do not need to understand how an API is developed; they just connect with other products and services over the interface. API use has exploded in popularity over the last decade, to the point that many of today's most popular online apps would not be feasible without them. [33]

3.5 Bluetooth Protocol

A common technology in everyday gadgets is the Bluetooth protocol. In modern times, most individuals have gained familiarity with using gadgets with Bluetooth technology. Bluetooth technology uses the publish and subscribe pattern[35]. The advantage of this pattern is low power consumption, its suitability with graphical interfaces and its popularity. Downsides are lack of scalability, message predictability, increased latency, and message delivery are not guaranteed[4].

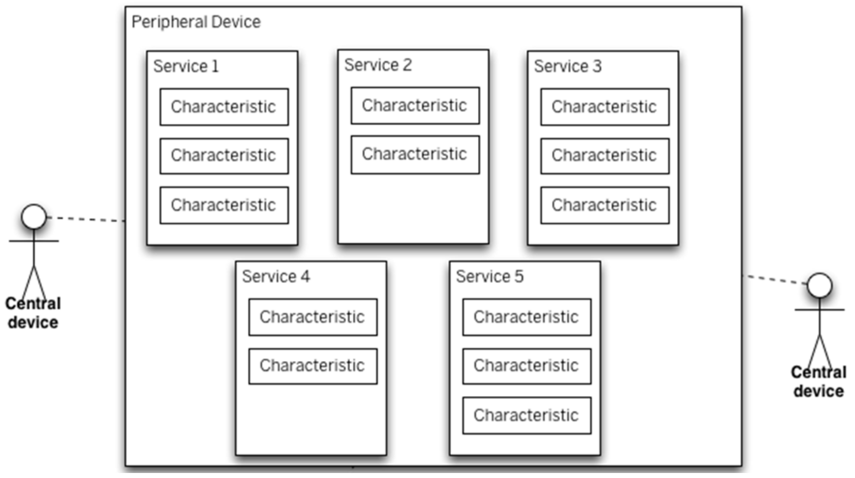


Fig. 3.6. This figure taken from the Arduino bluetooth documentation[35] shows a diagram of how the Bluetooth technology is built. An example of a central device is a smartphone, and an example of a peripheral device is an Arduino. For instance, in the Arduino script, it is possible to set up multiple services with characteristics that either publish or connect central devices to write values to the characteristic. An everyday example of writing values would be to change speaker volume or lights on or off using a smartphone application.

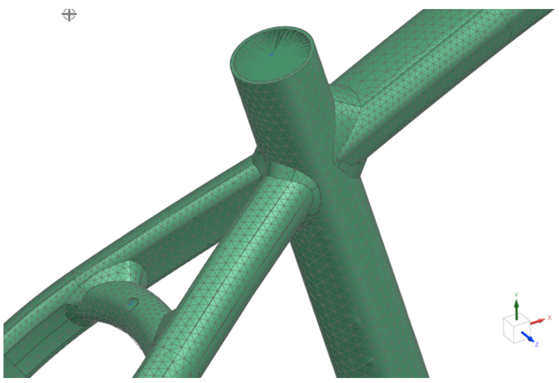
Chapter 4

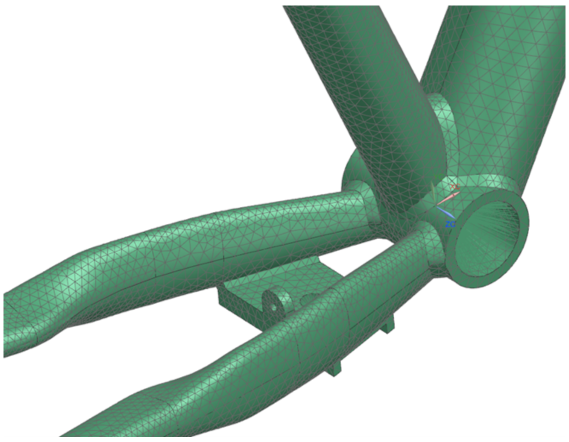
Description of the Structural Health Monitoring System

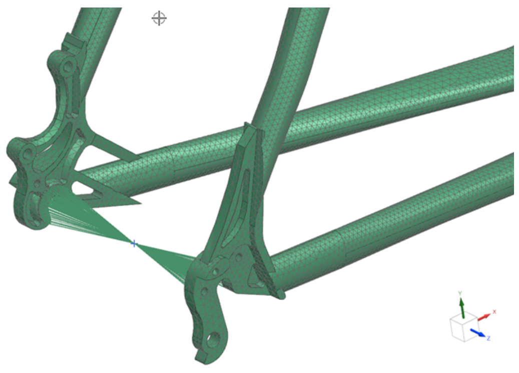
A comprehensive description of the whole SHM system, which was developed for the aim of monitoring the structural health of mountain bikes, will be provided in this chapter. The major goal of this study was to assess a digital twin system for Hardrocx bicycle frames that was capable of sensing loads and conducting SHM on the bicycle frames. Because to the need for a wide system bandwidth, comprehensive FEA or Functional Mockup Unit (FMU) were ruled out due to the high bandwidth requirements. The hardware and power supply available on a bicyle are also limited, and it was believed that a ROM would be the only solution. Despite the fact that these methods are most applicable to linear problems, bicycle frames are considered to react linearly to structural pressures until they collapse or cause substantial damage. In terms of software development processes, the system adheres to the theory given in sections 2.2, 3.3, 3.4, and 3.5 in order to implement the methodologies mentioned in section 3.2.

4.1 The Digital Twin FE Model

The manufacturer provided a physical and digital 3D model of a 19” Hardrocx SuperMotard M4 bike. The 3D model is an exact replica of the actual bike. Prior to meshing, only minor CAD elements that did not affect structural integrity were deleted. The bike model was idealized and meshed in NX using Teth 10 elements with a thickness of 6 mm. The FEM model comprises 154982 elements and 308957 nodes in total. Aluminum material attributes were attributed to the tetrahedral elements. RBE2 and RBE3 parts represented the seat and steering pin, as well as the wheel hubs. The meshed model was imported into FEDEM, where correct boundary conditions were given to free joints representing the front and rear wheel hubs. Following that, unit loads were applied at predetermined sites and directions to produce the ROM.


(a) Unit x-Load(1)


(b) Unit z-Load (2)


(c) Unit z-Load (3)

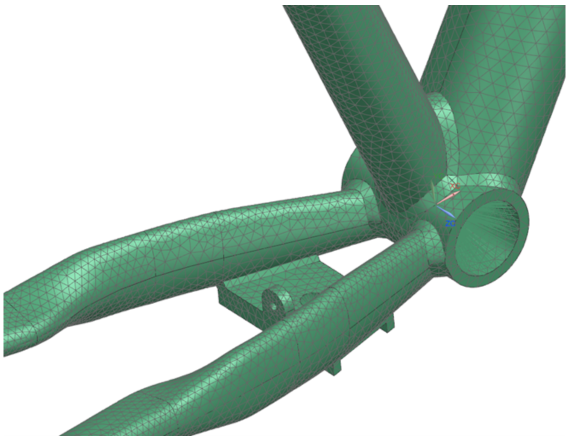

(d) Strain gauge #4

Fig. 4.1. FE model of the bicycle frame

4.2 The static ROM

Mendix presently does not enable co-simulation, and FE analysis of the whole frame is too CPU intensive to perform in real time. Furthermore, only stresses in specific [7] hotspots had to be computed. Because it provides a linear relationship between a few crucial input loads and output hotspot stresses/strains, this trade-off has shown to be an efficient strategy. [32]

Thus, a static ROM for the 19” Hardrocx bike was constructed by pre-computing the stresses at eight specified frame positions in FEDEM. These places were previously recognized as densely populated hotspots [23, 7]. Strain gauges on single legs placed in hotspots might then give stress time records for fatigue calculations.

Because of this, the static ROM consists of an 8 x 6 matrix (look-up table) including information on the load–stress relationship for the Hardrocx 19” frame. In order to get the stress distribution in the eight strain gauges, the inverse method is used to calculate the six structural loads, which are then multiplied by the 8x6 matrix.

In FEDEM, the graphic below depicts the stress contributions from each applied unit load. These stress distributions are too time-consuming to compute in real time, but they were utilized to determine the ideal strain gauge placements. The strain gauge findings are then computed in real time by multiplying the ROM matrix by the load vector determined by the inverse method.

Table 4.2 shows how the 8 gauge data corresponding to the 6 unique unit loads in Table 4.1 may be arranged into a matrix. Because the model is linear and solves for similar boundary conditions, the gauge stresses caused by any combination of unit loads given to the FEDEM model or multiplied by the ROM matrix provide identical results. As a result, any inversely determined combination of actual loads may be multiplied by the ROM matrix to estimate the real stress time histories. The static ROM is only useful when the stresses are less than yield. The different physical loads’ stress contributions may then be overlaid. Finally, stresses above yield will be detected by a smartphone app trigger event.

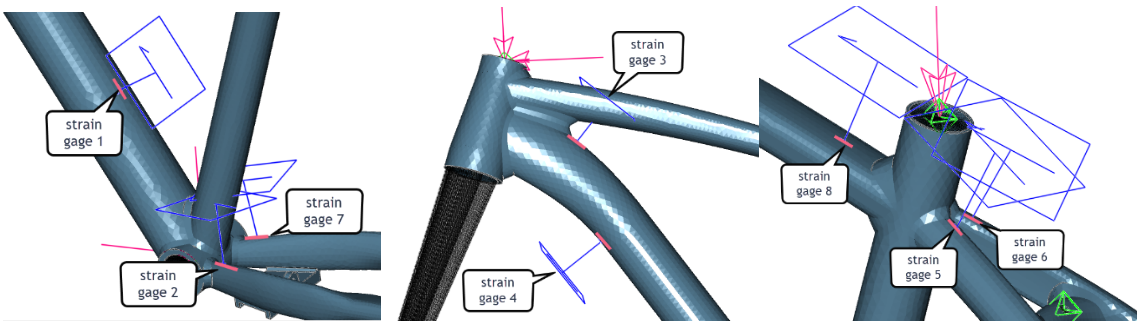
Unit-Load Matrix						
Stress-in Gauge-ID	Unit Load 1	Unit Load 2	Unit Load 3	Unit Load 4	Unit Load 5	Unit Load 6
1	2,62E+02	-4.15E+03	-1,25E+03	9,88E+03	-7,31E+02	4,60E+02
2	1,08E+04	3,20E+03	-2,32E+03	-1,87E+04	-1,85E+03	-3,95E+03
3	-1,38E+04	-2.89E+04	-6,02E+03	8,45E+04	-8,59E+03	4,04E+01
4	1,56E+04	2,56E+04	1,18E+04	-7,83E+04	1,64E+04	-9,89E+01
5	1,90E+04	-8,74E+03	-3,03E+04	2,41E+03	-2,05E+04	8,46E+02
6	1,87E+04	-8,82E+03	-3,03E+04	2,86E+03	-2,04E+04	9,35E+02
7	1,51E+04	4,34E+03	-3,47E+03	-2,58E+04	-3,23E+03	-4,76E+03
8	1,02E+04	2,76E+03	-1,98E+03	-1,70E+04	-4,83E+03	-2,26E+02

Table 4.1: The Unit Load Matrix

4.3 The Inverse method

Based on projected bicycle loads, many approaches may be used to compute bicycle stresses. Covill & Al [7] provide a thorough summary of the methodologies used to determine stresses induced by different load scenarios. They also identified road bumps at the front and rear wheels as the most crucial load instances, resulting in vertical accelerations. Brake forces are less significant, but they are quick to calculate, thus they are included in this research.

TrueLoads software [32] is a more current solution for capturing bicycle loads using strain gauges. The strain gauge distribution is optimized using unit loads applied to specific locations and directions on a physical asset's linear model. On TREK bikes, this technology is utilized to collect loads for frame design and optimization [32]. The favored method in this investigation was TrueLoads, however the hotspots were previously found in [23, 7], as shown in Figure 4.2. For final Digital Twin certification, real and virtual strain gauges were installed on these hotspots. TrueLoads would also need a costly Digital-to-Analog Converter (DAC) and at least 12 additional strain gauges to identify the six most essential loads illustrated in 4.3.



(a) Strain gauge 1, 2 & 7 **(b)** Strain gauge 3 & 4 **(c)** Strain gauge 5, 6 & 8

Fig. 4.2. Strain gauge distribution

As a result, the authors chose to estimate the frame inertia cyclist loads using data from a single low-cost Arduino IMU mounted on the baggage rack. Based on the recorded accelerations (a_x, a_y, a_z) and angular rates $(\dot{\alpha}_y, \dot{\beta}_y, \dot{\gamma}_y)$ around the global coordinate system shown in Figure 3.1, the most essential frame loads during a bicycle ride may be computed in real time, as shown in Table 4.1. As response forces in free joint springs, the vertical loads operating in the front and rear hubs due to acceleration and braking are incorporated. These are nonlinear compression springs that serve as tire models.

Because both front and rear wheel bumps contribute measurable vertical acceleration a_z, a vertical load distribution is estimated based on the bicycle geometry, and angular bicycle pitch acceleration $\ddot{\beta}_y$ is derived as the derivative of $\dot{\beta}_y$. When the pitch rate is negative, the front wheel is passing over a bump(wheel lift), and vice versa when the pitch rate is positive and the rear tire is travelling over a bump. Physical riding tests on the Hardrocx bike were used to evaluate the rider mass distribution (20% handlebar, 50% seat, and

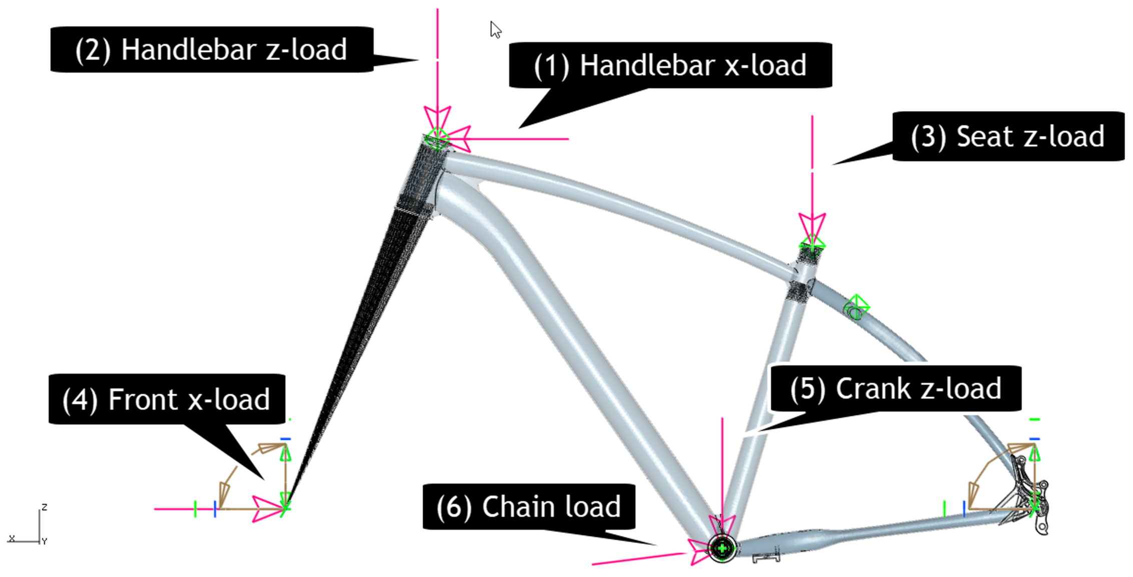


Fig. 4.3. Frame loads that have been applied

30% crank). Because weight distribution varies depending on riding position, all physical tests are conducted in the same sitting test position. Additional load cells on the seat and handlebar tubes would be required to record the rider mass distribution during varied sitting positions and offroad handling. This kind of force-driven digital twin arrangement is more precise, but it contradicts the intended simplicity and present budget limits.

The accelerations during front and rear wheel bump passes are calculated using the observed IMU pitch rate $\dot{\beta}_{yIMU}$ and vertical acceleration a_{zIMU}:

$$\text{Handlebar acceleration} \qquad a_{zH} = H\ddot{\beta}_{yIMU} + a_{zIMU} \tag{4.1}$$

$$\text{Crank acceleration} \qquad a_{zC} = C\ddot{\beta}_{yIMU} + a_{zIMU} \tag{4.2}$$

$$\text{Seat acceleration} \qquad a_{zS} = S\ddot{\beta}_{yIMU} + a_{zIMU} \tag{4.3}$$

$\ddot{\beta}_y = \frac{\dot{\beta}_{y,t+1} - \dot{\beta}_{y,t-1}}{2dt}$ (dt = sampling time increment) determines the pitch acceleration. These accelerations are response inputs to an inverse method capturing the most significant dynamic loads operating during a bicycle ride, as illustrated in 4.1. Future implementations will include inverse methods for collecting handling loads, however the existing hardware (Arduino shells) and instrumentation (one IMU) are insufficient for real-time offroad handling load computations. The true inverse method implementation, on the other hand, is more sophisticated and is tweaked by preliminary experiments.

The inverse method is based on simple yet rapid analytical calculations that estimate the primary vertical stresses and weight transfer during a bicycle ride's acceleration and braking. This response-driven method might be expanded to include managing loads. However, lightweight force transducers fitted on the handlebar and seat tubes can immediately sense the biker's sprung inertia loads, obviating the requirement for an inverse method. While the existing single IMU-based response-driven inverse method is rapid, it is confined to straight-forward bicycle riding and not to general offroad handling. While a force-driven digital twin is a more straightforward and accurate solution, the majority of force transducers are excessively hefty (see Figure 6.3) and will compromise the suggested SHM framework's simplicity.

Inverse Method					
ID	Force	Sensor Output	Property	Inertia Loads by Inverse Method	Description
1	Handlebar x-load	a_x	Mass	ma_x	Biker loads due to ax (mainly weight transfer during breaking) (100%)
2	Handlebar z-load	a_z	Mass	$0.2ma_{zH}$	Biker loads due to gravity and a_z (20% weight distribution)
3	Seat z-load	a_z	Mass	$0.5ma_{zH}$	Biker loads due to gravity and a_z (50% weight distribution)
4	Front x-load	a_x	Mass	$ma_{xIMU}, (a_{xIMU}<0)$ $0.02ma_{zIMU}, (a_{xIMU}>0)$	Front brake load (assume 100% of total brake load) or 2% rolling resistance
5	Crank load z	a_z	Mass	$0.3ma_{zC}$	Biker loads due to gravity and a_z (30% weight distribution)
6	Chain load	Torque	Pedal (radius)	Torque/radius	Applied compression load from crank torque (in lower arm)

Table 4.2: The Inverse Method

4.4 The Hardware

4.4.1 Arduino Microcontroller

The Arduino Uno WiFi Rev2 microcontroller is used to digitize the strains and accelerations that the bicycle frame and bicyclist are experiencing, as well as locally connect to a dedicated smart phone preferably attached on the bicycle handlebar. The code for making this happen can be found in C.1. Regarding the sensors, the sensors interface with the Arduino via two shields: the 9-axis motion shield[25] and the Crowtail-Base Shield[27], both of which are mounted on top of the microcontroller. It is necessary for this project to get acceleration in the a_x and a_z axes using the accelerometer, and the pitch in the y direction included into the 9-axis motion shield, which uses an BNO055 IMU. There is also an LSM6DS3 IMU installed directly on the Arduino Uno WiFi Rev2 used for benchmark purposes. The crowtail shield is used to connect the strain gauge sensor modules[26] without the need for solder; figure 4.11 illustrates this. All of the components in this system run at 5 volts, have an acceptable bit resolution, and are well documented.

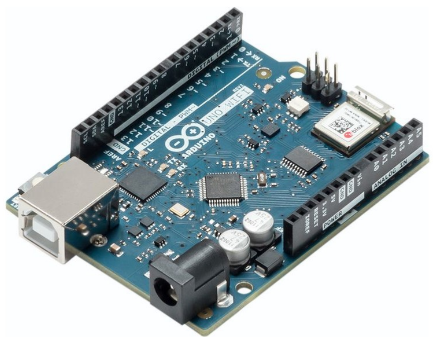


Fig. 4.4. Arduino Uno WiFi Rev2

An overview of the different states of the Arduino program is pictured in 4.7. The arduino script is setup to read sensor values once every ten milliseconds, resulting in a sample frequency of one hundred hertz. Accelerations, strain gauge and timer values are stored in two data types, *strain_gauge* and *acc_gyro*, which are customized datatype created specifically for sending the sensor data via Bluetooth to the phone. These data types are 16 and 10 bytes in length, respectively, which is inside the 20-byte payload limitations of the Bluetooth version running on the Arduino. The code for the data type can be found in appendix C.1. The union type allows for the storing of several data types, in this case struct and byte, in a single data structure. An unsigned long of 4 bytes (32 bits) is used to store the trip timer, and six short data types of 2 bytes (16 bits) each are used to store sensor data from the IMU and strain gauges.

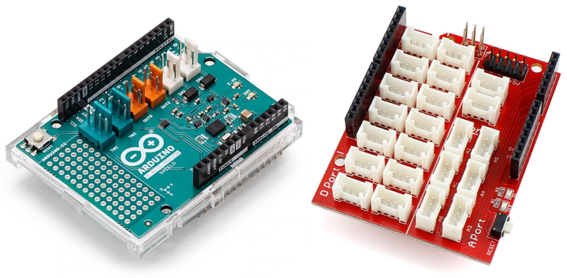


Fig. 4.5. Arduino shields: 9-axis left, crowtail right

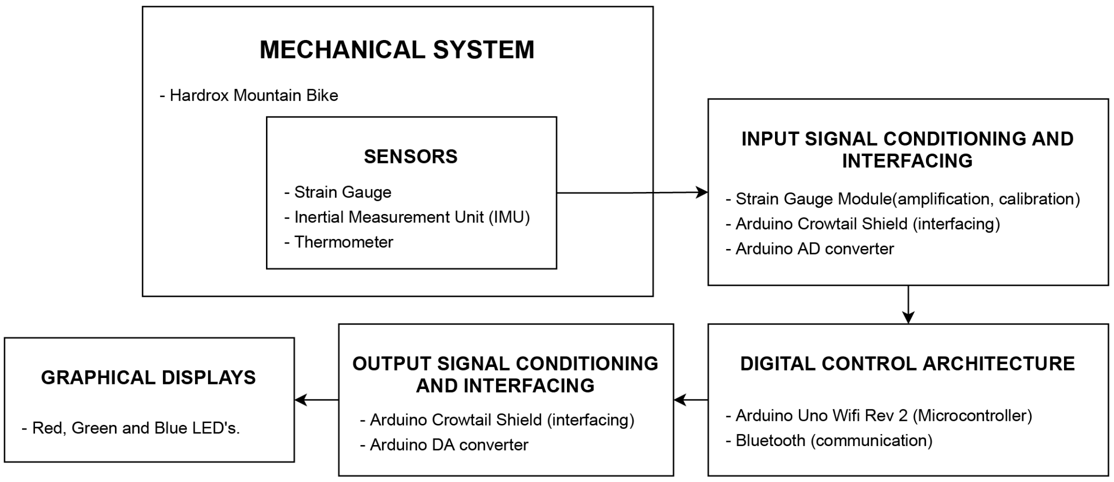

MECHANICAL SYSTEM
- Hardrox Mountain Bike
SENSORS
- Strain Gauge
- Inertial Measurement Unit (IMU)
- Thermometer
INPUT SIGNAL CONDITIONING AND INTERFACING
- Strain Gauge Module(amplification, calibration)
- Arduino Crowtail Shield (interfacing)
- Arduino AD converter
DIGITAL CONTROL ARCHITECTURE
- Arduino Uno Wifi Rev 2 (Microcontroller)
- Bluetooth (communication)
OUTPUT SIGNAL CONDITIONING AND INTERFACING
- Arduino Crowtail Shield (interfacing)
- Arduino DA converter
GRAPHICAL DISPLAYS
- Red, Green and Blue LED's.

Fig. 4.6. Overview of the mechatronics system.

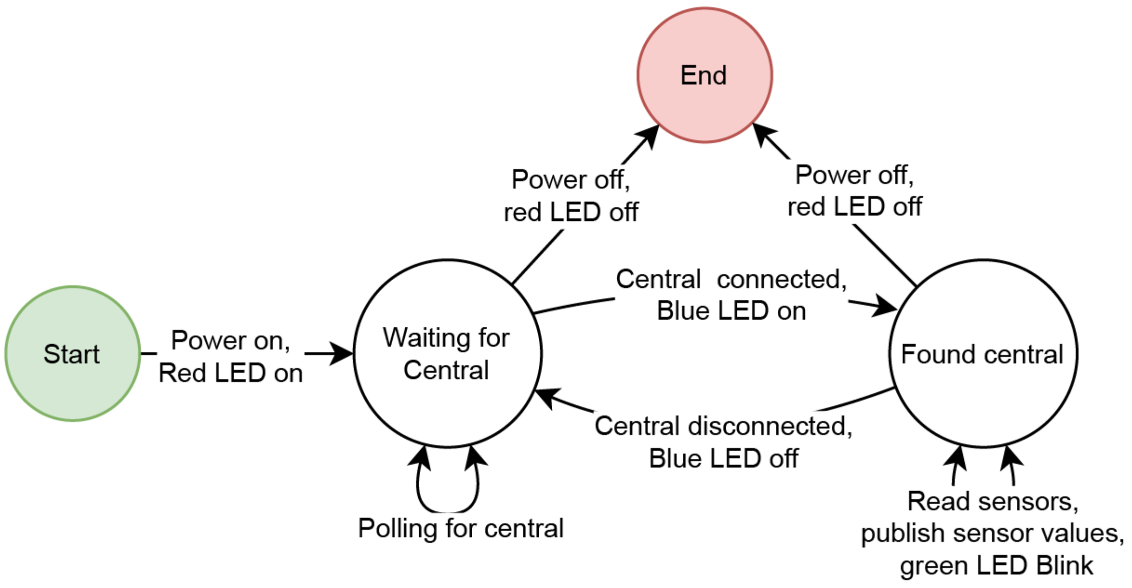
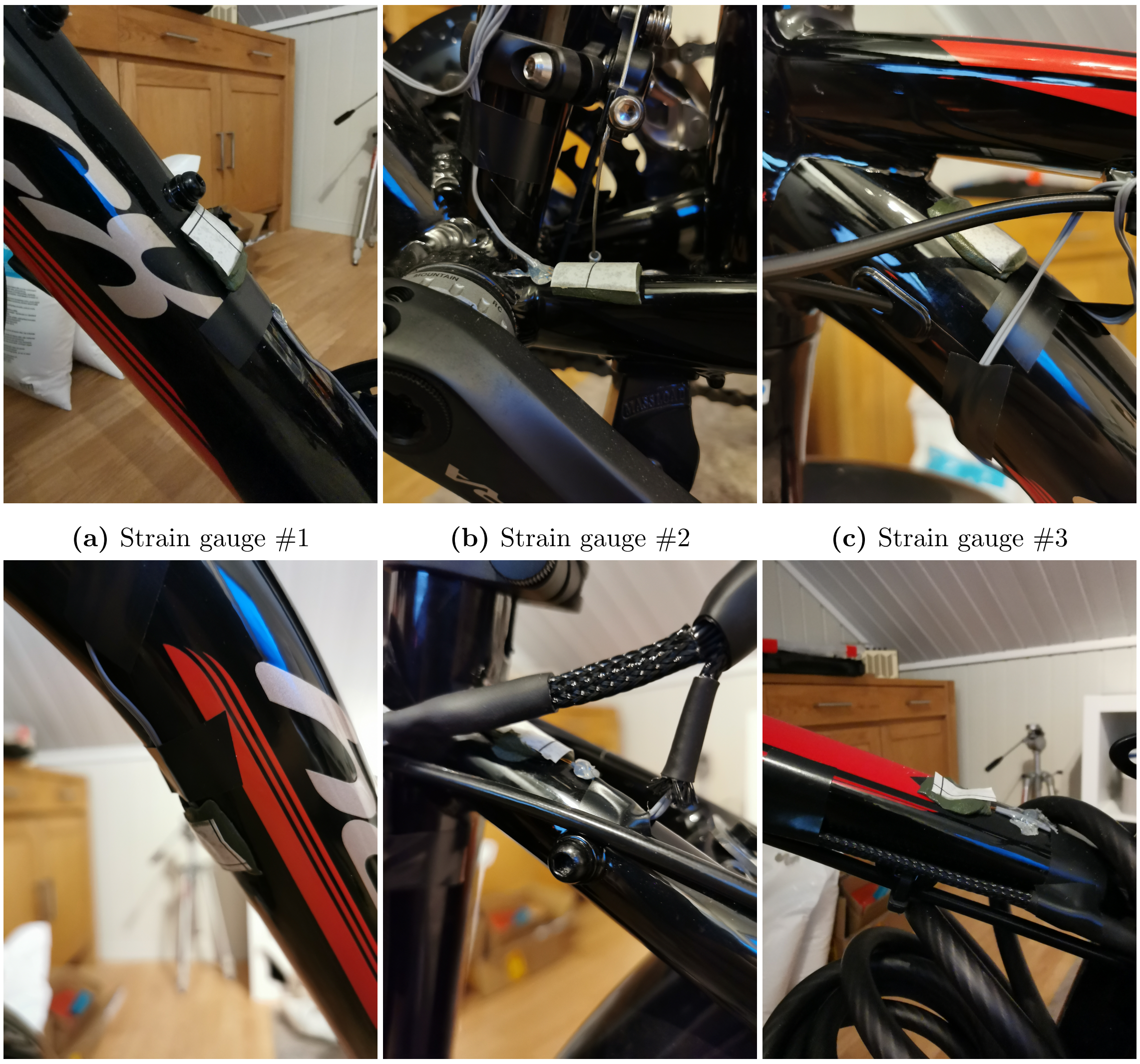


Fig. 4.7. Arduino state diagram to describe the Arduino script found in C.1.

4.4.2 Strain gauge and instrumentation

The strain gauges used are of the type FLA-w-350-11-1L, a general-purpose metals strain gauge developed by Tokyo Measuring Instruments Lab[36]. Each area for the strain gauges was roughened using 220 grit sandpaper and cleaned with acetone. The strain gauge was very carefully attached to the frame using a sheet of teflon, which is chemically resistant to the specified adhesive. After the glue had dried, a squared piece of HBM's ABM75 covering material was gently put on top of the strain gauge to establish a connection between the gauge strain and the frame and to secure the strain gauge. A few further efforts were taken to secure the area and eliminate as much noise as possible. Several wires were attached to the frame near the covering material. A section next to the strain gauge was bonded to the wires that were especially exposed. Additionally, the wires were put into cable hoses and heat shrink tubes were added at entrance locations and where the cable hoses needed to be stretched.



(a) Strain gauge #1 **(b)** Strain gauge #2 **(c)** Strain gauge #3

(d) Strain gauge #4 **(e)** Strain gauge #5 **(f)** Strain gauge #8

Fig. 4.8. Strain Gauge Physical Locations.

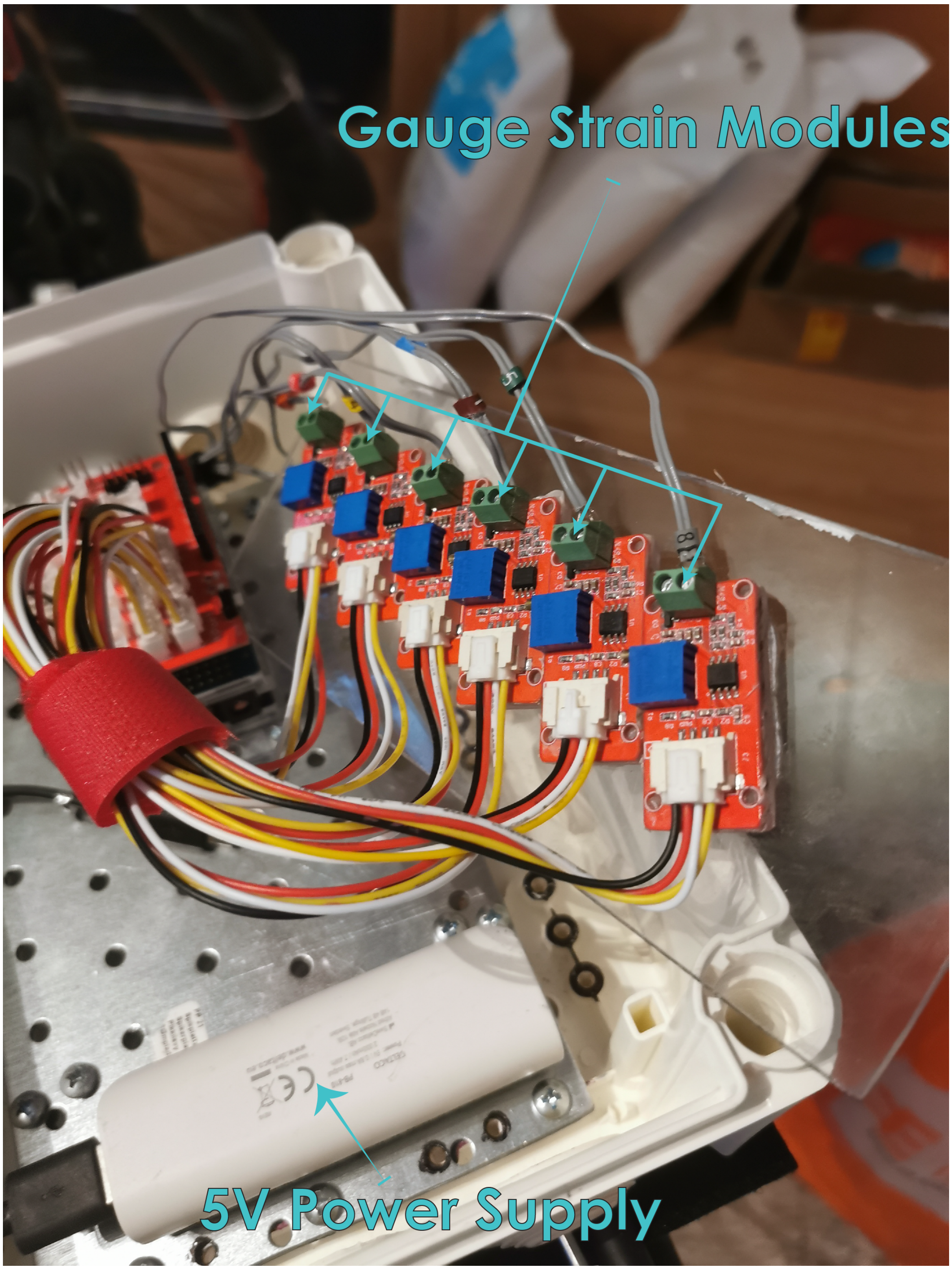


Fig. 4.9. Strain Gauge Modules and 5V power supply with the system turned off.

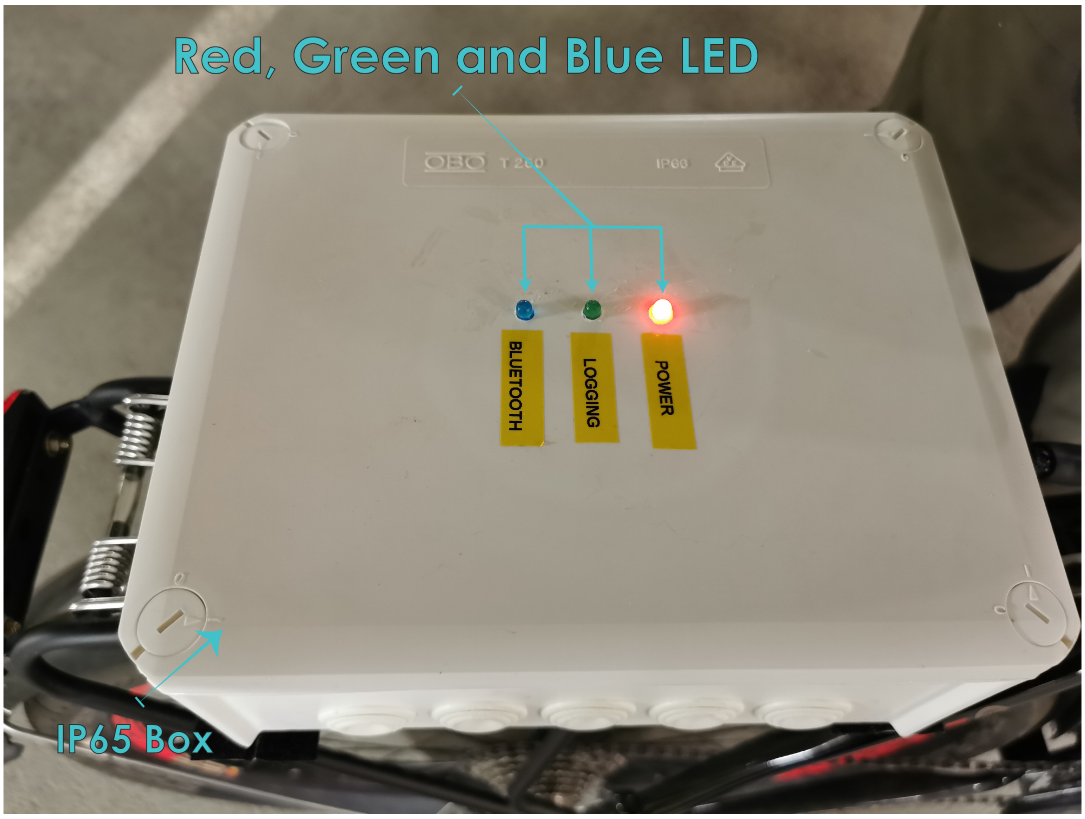


Fig. 4.10. Box closed with the system turned on.

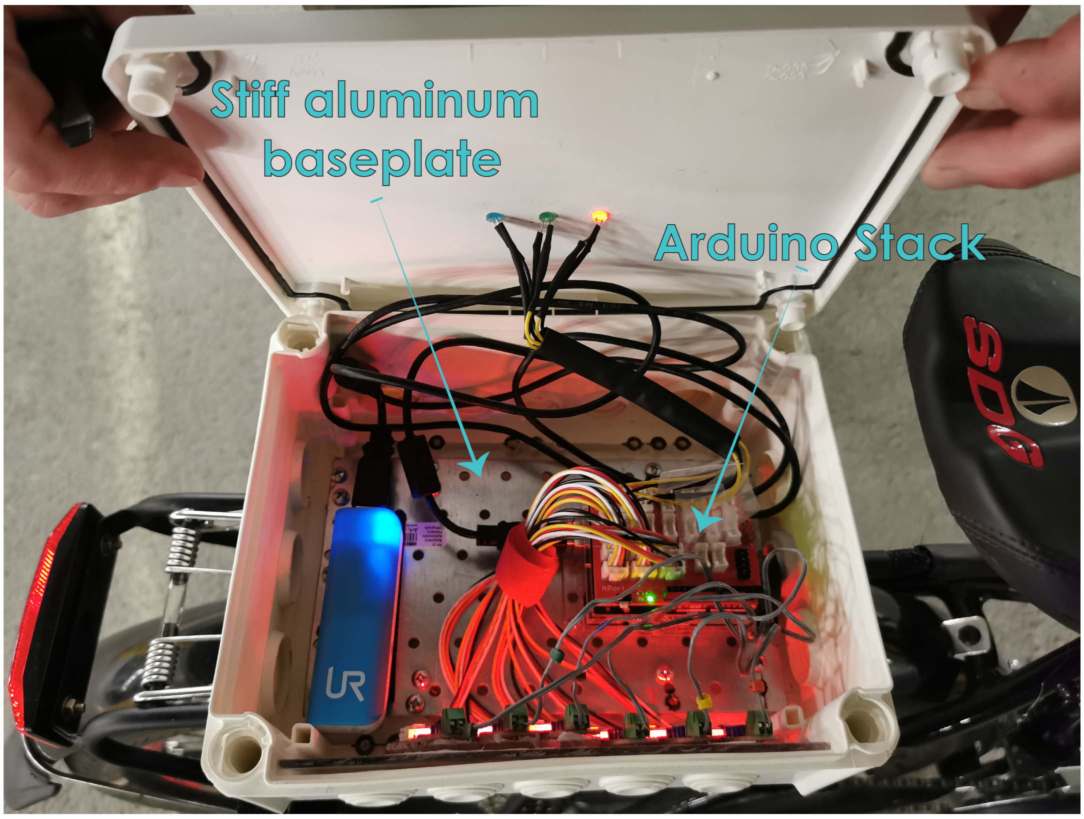


Fig. 4.11. Box open showing the baseplate and arduino stack with the system turned on.

4.5 The Software Solution

The architecture for the software solution follows the theory regarding the IoT and IIoT differences, IoT framework, IoT platform, IoT protocols presented in chapter 2 in order to create a system that is able to: Solve problems, interconnect with other technology, create an ecosystem of producers and consumers, reduce complexity with IoT setup and operation, enhance modifiability, dependability and maintainability, and accelerate the commercialization of IoT products. The theory section 3.3 is also present.

Be assured that the system is still in its infancy, but the core of the system is complete. The framework of the system is built upon the Mendix platform which utilizes TCP connection to connect to any server using Transmission Control Protocol (TCP)and HTTP, as well as Bluetooth network communication to communicate between nearby devices. Since the whole solution is highly modifiable and can be interconnected with other systems using various API's suppored by Mendix, the solution can be labeled both as an IoT framework, and also an IoT Platform.

The ecosystem developed using the Mendix platform is shown in figure 4.13. The Arduino microcontroller is the system's producer, since it gathers raw sensor data. The raw sensor data is sent over LAN to a linked smart phone through the Bluetooth protocol. The smartphone consumes the data and sends it to the server over RESTful HTTP and TCP. Once data is committed to the server, a Mendix task queue dedicated for doing multi-threaded work, will be notified and thus consume the data as well as generate new information using the inverse method and ROM described in sections 4.3 and 4.2.

To facilitate the setup and operation of the IoT asset, the system is designed in such a manner that it requires little programming experience to maintain or support other devices, standards, and so on. Additionally, adding new assets that the platform user can monitor is a breeze. Mendix's cloud is also exceptionally available due to its Cloud Foundry and AWS architecture.

Figure 4.12 is created to exemplify the pipeline used to enrich the data. Note that in the current system, there is no AI or ML applications running, but the system does support it by using the Java virtual machine.

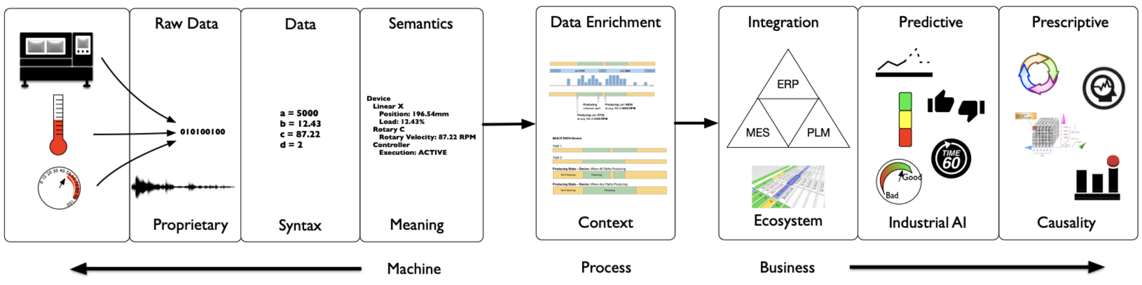


Fig. 4.12. The figure illustrates the steps required to turn raw data into valuable information for decision-making processes. Take note that the data enrichment process is what enables these decisions, and the intention of the process is to provide context.

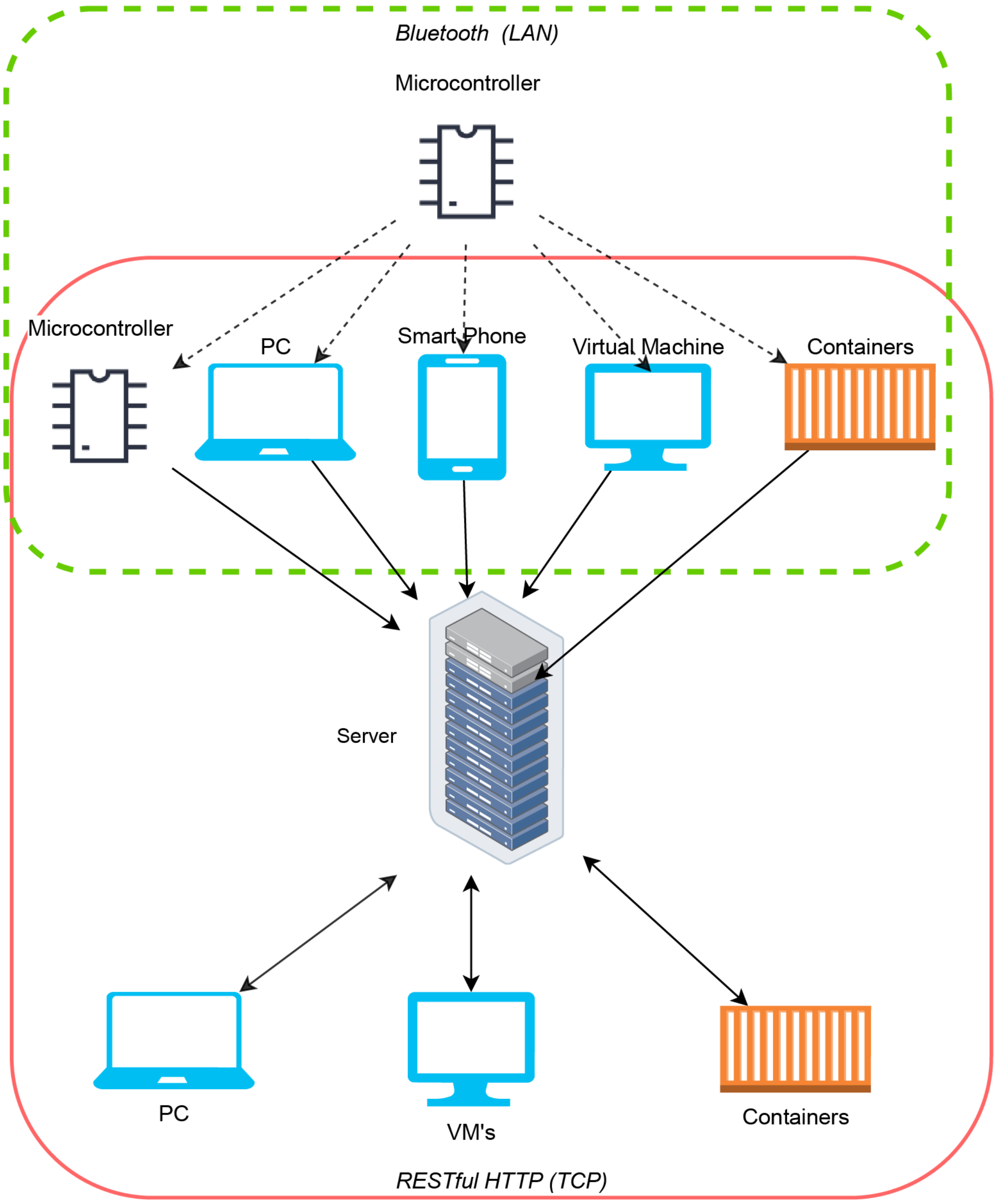


Fig. 4.13. The figure depicts the IoT ecosystem created by the technologies used in this system. Local connection is shown by the green dotted boundary, whereas TCP client-server connectivity is indicated by the red solid line. Numerous devices may be connected to the system using the Mendix API's. The diagram depicts a simplified version of the whole system, notably the server solution, which is handled automatically by the Mendix system and has various components that the author did not configure.

4.5.1 The Mendix Client Application

In order to connect the bicycle instrumented with sensors to the ecosystem, the Mendix client plays an essential role. Throughout the bicycle ride, the Mendix client is accessible via a web browser on essentially any device, but is suggested to be used on a smart phone during ride or computer during development and testing. During the ride, the smart phone is supposed to be mounted to the handlebar of the bicycle. By logging in and navigating the site, the user may create a bicycle riding event, delete existing assets, configure existing assets, check the data, and save the data in Excel or Comma-Separated Values (CSV) format. At the end of the document, appendix D presents a total of fourteen pages which can be studied in order to get a glance of the whole system. The functionality of the Mendix edge client varies depending on who uses the web site. For instance, an administrator has added functionality since the role is essential to setup the whole system. Moreover, other security measurements taken are restrictions put on each user's ability to create, modify, or delete particular data on the server. Using Mendix Studio Pro's built-in documentation features, which can be found in E, the author has documented the whole solution. Mendix applications are completely reliant on the Mendix developer's ability. There are various difficulties in designing such an application, and the learning curve is initially somewhat high, but the system is well documented, notifies the developer of errors, and allows for debugging.

Database

Mendix Studio Pro 9.5.0's domain model can be observed in figure 4.14. The Domain Model is used to create the database for the Mendix application that the ecosystem and IoT platform is all about. For NTNU students modeling databases, the process is akin to building ER diagrams, which is a well-known approach for producing Entity Relationship (ER) diagrams. The rounded blue squares in the figure represent entities. The arrows that connects the blue squares symbolize one-to-one, one-to-many, and many-to-many connections, depending on the situation. Mendix creates database tables for entities and relations in a fully automated manner. The supported data types are seen to the right of figure 4.14. All supported data types inherit from the Data entity. The DataType entity is only possible to be deleted and created by an administrator.

In the database, the Asset is one of the most essential entities to understand. Each asset necessitates the use of one or more Bluetooth-compatible devices (e.g., Arduino). Also, every device should offer one or more Bluetooth services to locally connect the devices together. Furthermore, each Bluetooth Service should have at least one Characteristic that indicates a data type that corresponds to an entity in the domain model. There is no limit to how many Bluetooth characteristics a Bluetooth service may have. A one-to-many relationship exists between the Event and the Asset, Device, Service, Characteristic, and EventBatch entities, which means that events may be associated with a variety of distinct

Assets, Devices, Services, Characteristics, and EventBatch entities. The Data entity is also connected with the Event, enabling other Bluetooth data entities (strain_gauge, acc_gyro, inverse_data, uint, or float32) to inherit the connection, leading in a simpler model, database, and less associations necessary for any data types created in the future. It is possible to cast the Data entity to one of the child entities in a microflow to check which datatype that is associated to a given event. The red squares around the relationships show that the database will cascade after deletion. In other words, when an Asset is removed, all connected Devices, Services, and Characteristics, etc. are also erased, guaranteeing that the database does not hold any unnecessary information.

User Interface

The User Interface (UI) is created in Mendix Studio 9.5.0 utilizing pages. In pages, the user interface is developed using the What-you-see-is-what-you-get (WYSIWYG) manner, which involves dragging and dropping widgets that add functionality to the page. These functions include website navigation, data visualizations, data maps, and user login. The application consists of fifteen pages in total. These sections enable users to add and update assets, devices, and events, as well as see real-time sensor data. Additionally, the sites are created with consistency in mind to ensure that graphical components and functionality are consistent across all pages. The following pages in D provide designs for the pages and a description of the implemented functionality.

4.5.2 System performance

Previously, there were several concerns with missing data owing to the high volume of internet traffic produced by the use of 100 hertz. Now, when monitoring a single Bluetooth feature, the software is capable of maintaining a continuous 100 hertz operating frequency. The prior issue was fixed by reducing the number of server transactions by transmitting data in chunks, which reduced the number of server transactions. To begin, the Mendix edge client connects to the Arduino's data through Bluetooth and subscribes to it. As a response, Arduino publishes the data at a frequency of 100 hz to the subscribers. It is then appended to a string in the Mendix Edge Client, which is sent to the server every 250 milliseconds. The data string is processed on the server side and then committed to the database along with the sensor values and timestamps. See B.2 for the Java code run on the server, and A.1, A.2, A.3 for the JavaScript code run on the client.

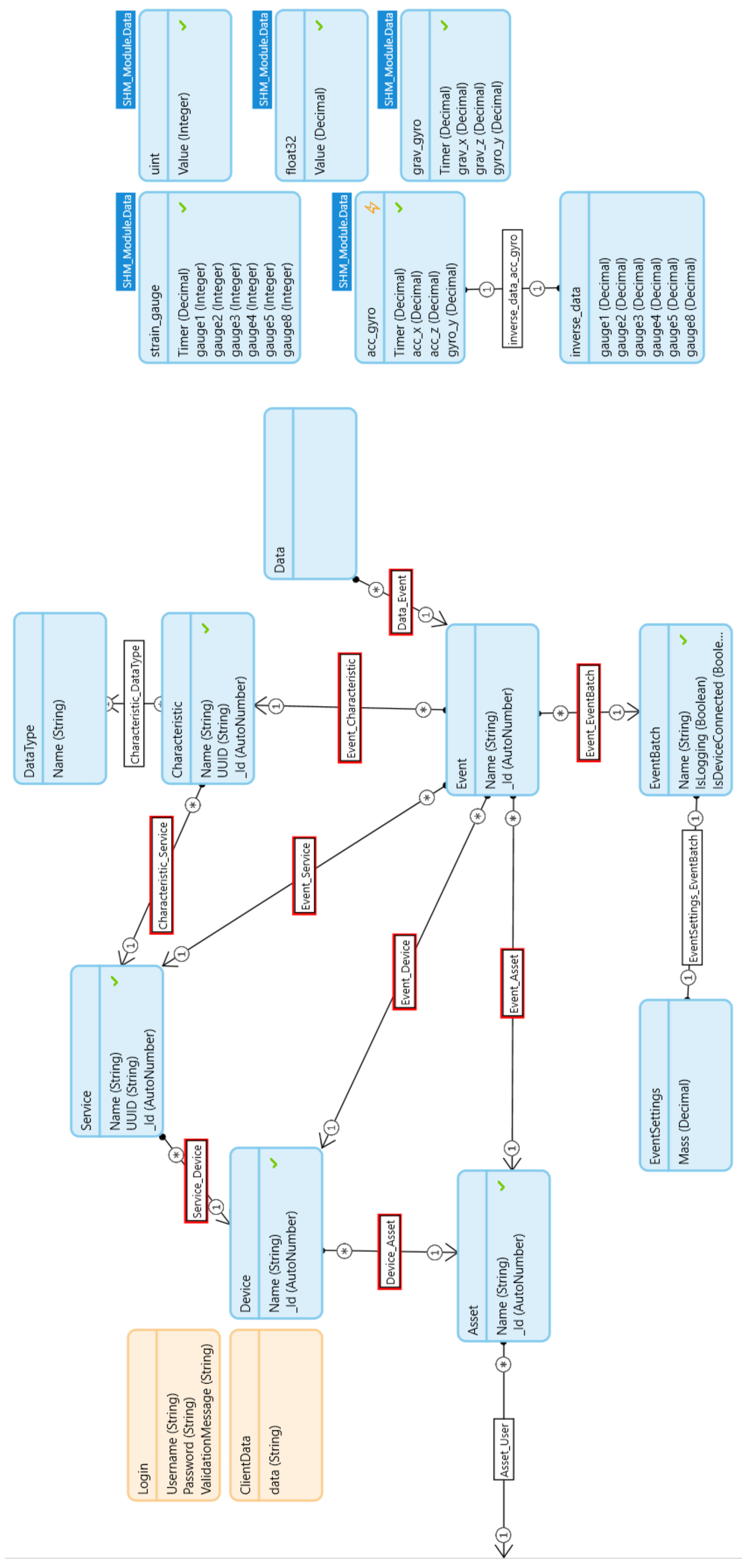


Fig. 4.14. The Domain Model

4.5.3 System modifiability

The Mendix application is highly modifiable since it is developed using the Mendix PaaS. The application is easily adjusted in the future by someone without any previous experience in Java or JavaScript. The development tools shown in Figure 4.15 were utilized throughout the development stage and should be used in the future while developing on the specified system.

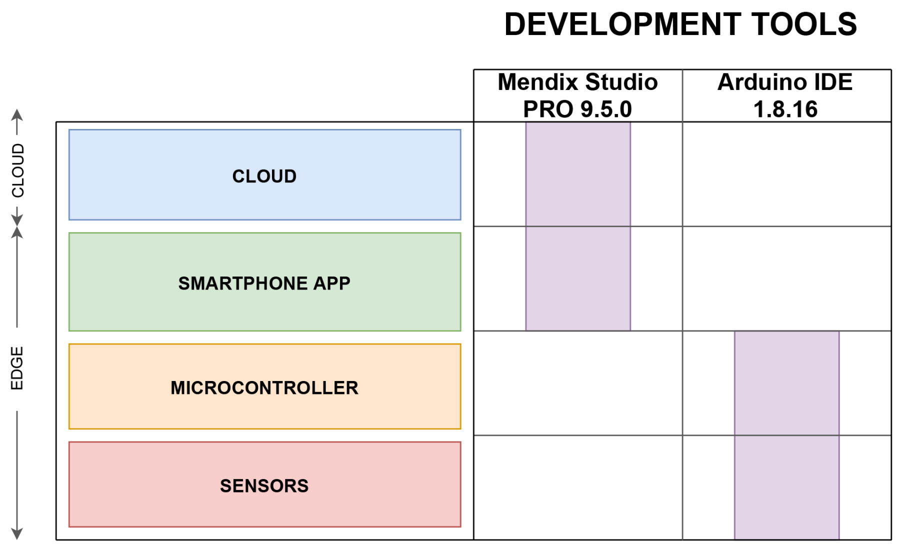


Fig. 4.15. Overview of the tools used for development.

Chapter 5

Recording Calibration and Setup

This chapter will detail the procedures for configuring the sensors. The system collects data from two distinct types of sensors: an IMU sensor (BNO055) and six strain gauge sensors. To produce useable data from the sensors, one procedure has to be performed for each sensor. This chapter details these activities.

5.1 Strain Gauge Sensors

5.1.1 Step 1: Adjusting potentiometer

A total of six strain gauges have been installed on the bicycle. Each sensor is connected to a separate module which can be seen in 5.3, which is used to ensure that it is properly calibrated. The values that can be collected by the AD converter on the microcontroller are limited to those that fall between 0 and 1000. Therefore, each sensor had to be calibrated within this range, and thus the range of strain measured in the frame had to be within this range as well, otherwise the system would fail. It was necessary to submit each sensor to the largest load possible in order to get an adequate value for each sensor. As a result of a high load, the different strain gauge sensor values were ranging between -100 to 200. As an outcome of these observations, the value of each sensor was adjusted to a value of 500. In this case, the highest value is estimated to be around 700, and the lowest 400.

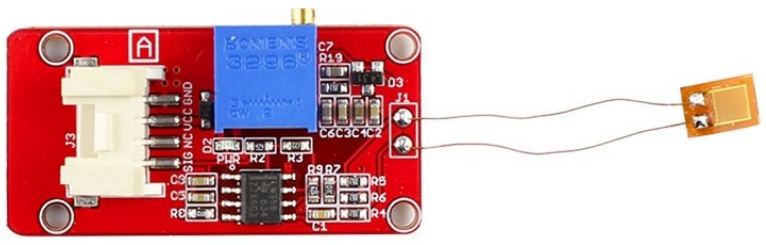


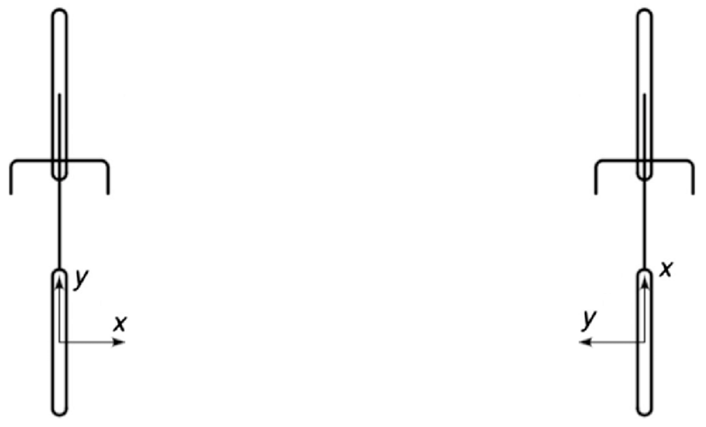
Fig. 5.1. Crowtail strain gauge module. The variable resistor is colored in blue.

5.2 IMU

The system is currently equipped with two different IMU's; BNO055 and LSM6DS3. This section only concerns the calibration of the BNO055.

5.2.1 Adjusting the global x-y-z direction

Figure 5.2a exemplifies how the IMU sensor used in the system is mounted on the bicycle. The BNO055 is spun clockwise by 90 degrees which makes the x-axis point in the opposite direction when compared to the y-direction used in the inverse method coordinate system as displayed in 5.2b. As a result, the value read from the gyroscope in the BNO055 will have the opposite sign. This has been accounted for while programming the inverse method in java, as well as post-processing the data in excel. This is perfectly acceptable and will have no influence on the values needed by the selected DT methods.



(a) IMU coordinate system. **(b)** Inverse method coordinate system.

Fig. 5.2. The different coordinate systems used in the IMU sensor and the inverse method calculation.

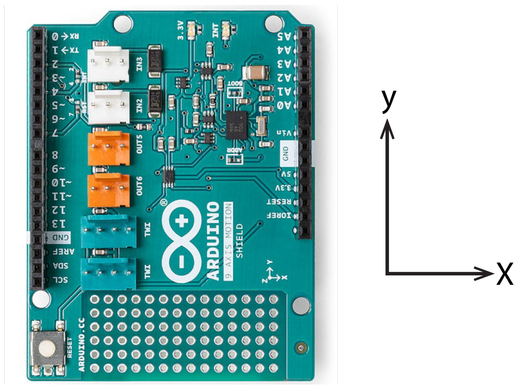


Fig. 5.3. 9-axis motion shield coordinates.

Chapter 6

Test Results

The purpose of this chapter is to conduct a series of tests to determine whether or not the concepts and methodologies chosen have been executed correctly. This series consists of three exams. The initial step is to validate the digital twin methods' correctness via the collection of data from a load cell. The second test is focused with system performance and is designed to determine the overall speed of the system. Test number three demonstrates how easy it is to add and monitor a new device to the system. The fourth and final test conerns the measurements taken from three bicycle rides

6.1 Digital Twin Validation

For the purpose of determining the relationship between strain and force acting on the bicycle seat, a load cell was used as shown in 6.1. With this configuration, the aim is to capture the most exact data possible in order to validate the ROM. The ROM presented in Table 4.1 is valid only if the bicycle frame exhibits linear behavior. As a result, a physical test was undertaken to compare the physical and digital bicycle models. Constructing a connection between the load cell and a Spider 8[31] strain gauge measuring equipment made it possible to record the data in catman software [30]. The weight was placed on the load cell with the help of a forklift, as shown in 6.2. As indicated in Figure 6.3, the seat tube was replaced with a force sensor installed on an equivalent aluminum tube and 200 kg was applied with the forklift.

As indicated in figure 6.4, the output stresses from the physical test were recorded and compared to the simulated gauge stresses. Correlation is excellent, and the right graph illustrates the linear relationship between simulated and observed stresses. This demonstrates that the digital twin FEDEM model accurately predicts real-world physical stresses and that, owing to the linear structural behavior, the FEDEM model can be represented by a static ROM.

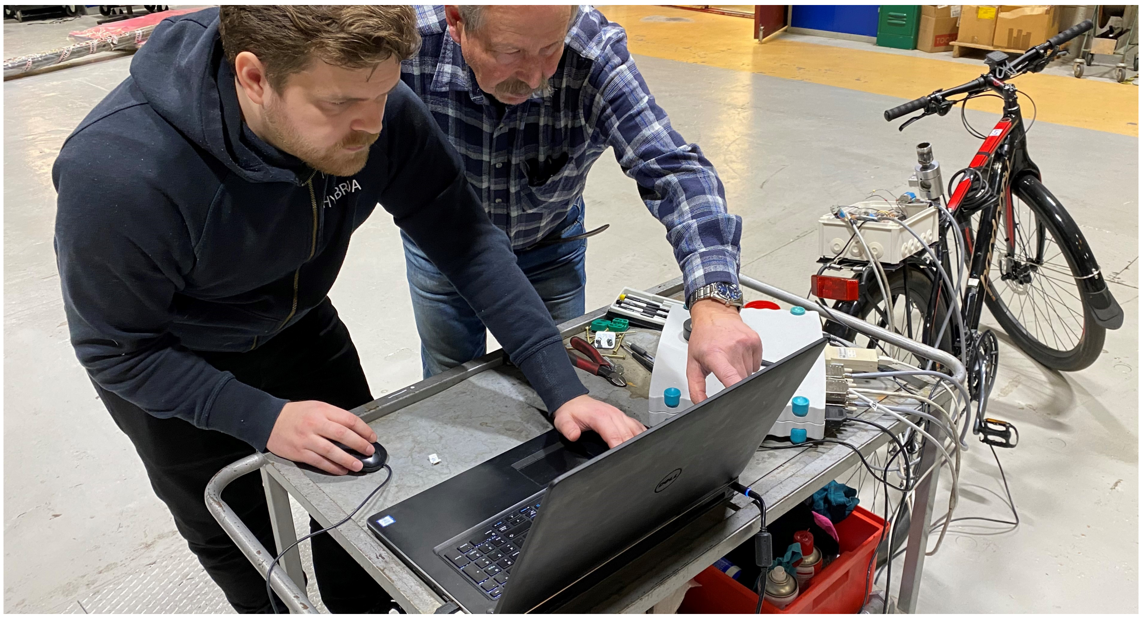


Fig. 6.1. Load cell is located at the seat of the bike. The data from the load cell and strain gauge sensors are read in catman.

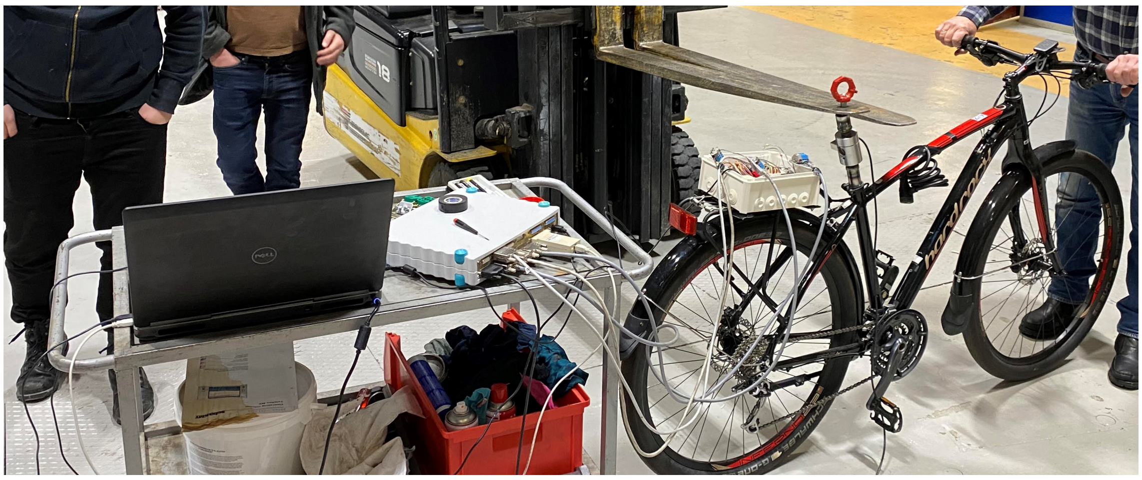


Fig. 6.2. Placing weight onto the bicycle seat using a forklift.

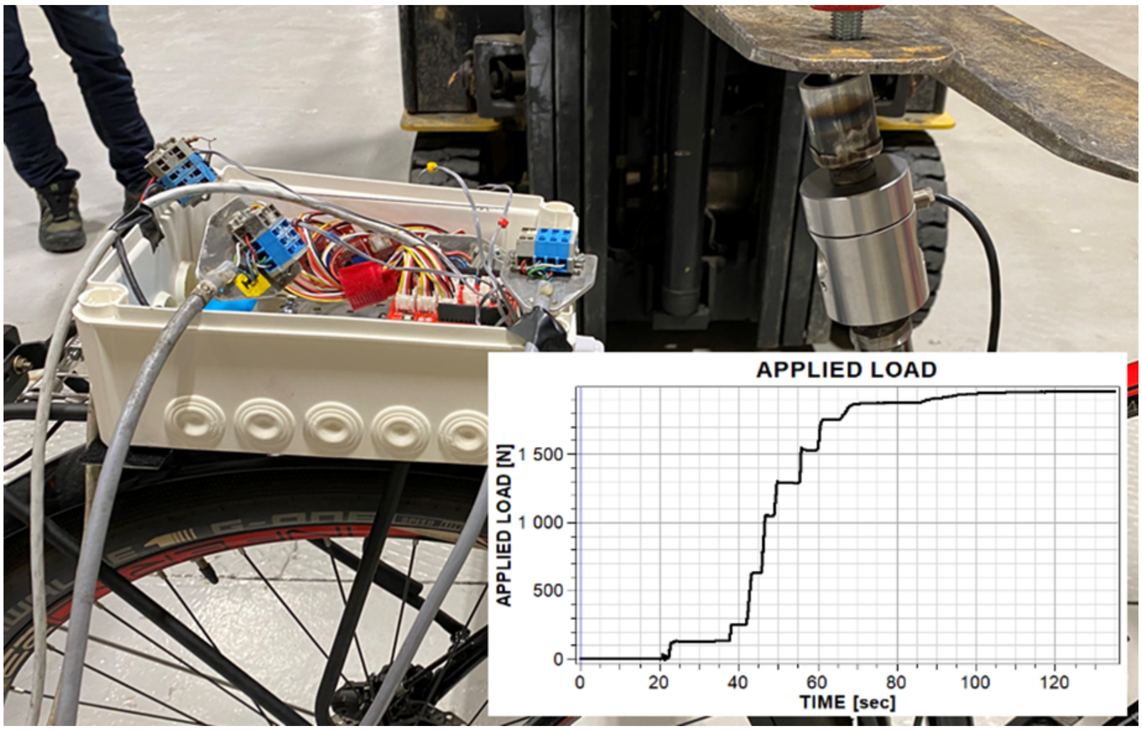
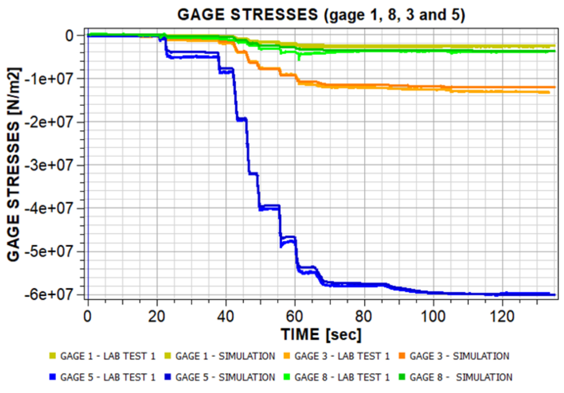
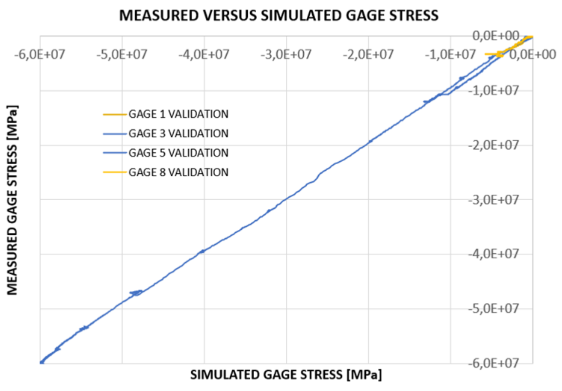


Fig. 6.3. Physical benchmark of the Digital Twin.



(a) Gauge stresses



(b) Simulated gauge stress

Fig. 6.4. Simulated versus measured gauge stresses

6.2 Testing the performance of the system

6.2.1 Monitoring single values

As shown in tables 6.1-6.4, the sensor data is monitored at approximately 100 hertz sampling rate. Each characteristic were monitored in over three different time-spans, 15s, 30s, and 60s. One can observe that the LSM IMU sensor slightly outperforms the BNO055 IMU sensor.

Event Name	Hertz
SG 15	99,64
SG 30	99,61
SG 60	99,54
Averaged	99,60

Table 6.1: strain_gauge

Event Name	Hertz
BN 15	98,34
BN 30	98,97
BN 60	99,04
Averaged	98,78

Table 6.2: acc_gyro using the BNO055.

Event Name	Hertz
LSM 15	99,64
LSM 30	99,58
LSM 60	99,59
Averaged	99,60

Table 6.3: acc_gyro using the LSM.

Event Name	Hertz
GG 15	99,22
GG 30	99,14
GG 60	99,11
Averaged	99,16

Table 6.4: grav_gyro using the LSM.

6.2.2 Monitoring Multiple Values

When monitoring several values at the same time, the sample frequency declines from a constant 100 hertz to about 74 hertz, as seen in table 6.5 with two recorded characteristics, and 60 hertz in table 6.6. The value indicates that a greater number of subscribed features will result in a lower sample frequency.

Event Name	Hertz (strain_gauge)	Hertz (acc_gyro)
SG BN 15	74,05	73,86
SG BN 30	72,94	72,83
SG BN 60	73,69	73,78
Averaged	73,56	73,49

Table 6.5: Logging two characteristics: strain_gauge and the acc_gyro using the BNO055 IMU.

Event Name	Hertz (strain_gauge)	Hertz (acc_gyro)	Hertz (grav_gyro)
SG BN 15	60,42	59,82	59,30
SG BN 30	59,90	60,48	60,22
SG BN 60	59,84	59,90	59,81
Averaged	60,05	60,06	59,77

Table 6.6: Logging three characteristics, the strain_gauge, acc_gyro and grav_gyro using the BNO055 IMU.

6.3 Testing the modifiability of the system

6.3.1 Introducing a new 32-bit float datatype into the system

This situation will arise if a user is unable to access a certain data type. For example, if a brand-new sensor is used that necessitates the use of a distinct data type, it is possible that this data type will not be supported by an existing data type. Therefore, this test illustrates how the modifiability specified in 3.3 makes it straightforward and affordable to add new data types to the set of supported data types. As shown in 6.7, the process of adding a new data type took in total 4 minutes.

Introducing a new 32-bit float datatype into the system		
Step	Description	Time
1	Updating the Domain Model	1 min
2	Updating the Javascript Code	1 min
3	Testing	2 min

Table 6.7: Process for adding a new 32-bit float datatype to the system.

6.3.2 Adding a new Bicycle asset

With this test, the goal is to show how late binding may be utilized to speed up the process of adding a new asset to a project. According to 6.8, the total duration for creating a new Asset was 200 seconds. The stages in this method presuppose that the Bluetooth service- and characteristic has previously been identified and that the datatype is supported by the system, as described in table 6.7.

It is crucial to recall that step 6 leverages late binding, which means that the datatype for the characteristic is determined at this step. The administrator will create this data type, which will be stored on the server and retrieved once the data string containing the recorded sensor readings arrives. When the data arrives at the server, it is processed and placed in the entity with the same name in the domain model 4.14. This again increases usability since it requires fewer steps to select the data type for the characteristics when existing data types will be reused.

Adding a new asset to the system and configuring it.		
Step	Description	Time
1	Uploading Arduino Code to Microcontroller	60 s
2	Logging into the SHM Webpage	10 s
3	Create and configure the asset	20 s
4	Create and configure the device	20 s
5	Create and configure the Services	30 s
6	Create and configure the characteristics	60 s

Table 6.8: The process of setting up and adding a new asset to the system.

6.4 Measurements taken from three bicycle rides

Initial testing traveling over the conventional speed bump shown in Figure 6.5 at a speed of 20 [km/h] demonstrated that a greater sample rate higher than 100 hertz was required to capture the dynamics triggered by the combination of stiff tires and transient impact loads upon striking the hump. Due to the sample rate available when recording all eight strain gauges and IMU sensors, smoother test rides, as seen in Figure 6.6, were chosen. The Bromstadekra road chosen has a rough surface and one of Trondheim's highest speed bumps.

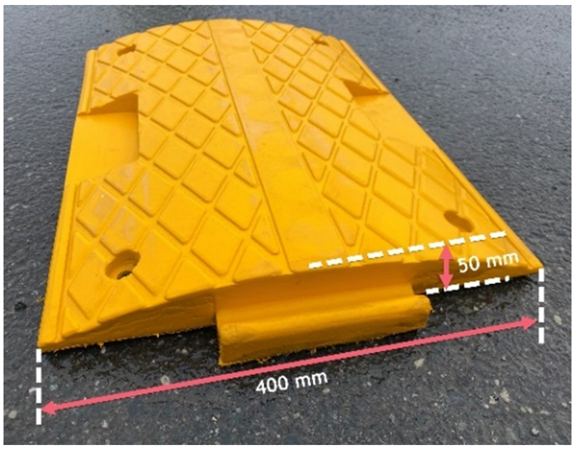
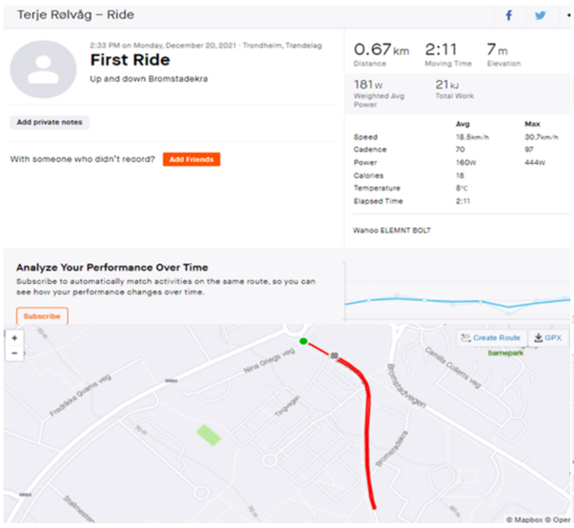


Fig. 6.5. Standard speed bump

Vertical a a_{zIMU} and longitudinal a_{xIMU} accelerations in the range of 0-2.5G and pitch rates $\dot{\beta}_{yIMU}$ up to 60 rad/sec were recorded by the IMU sensors, which were utilized by the inverse method to compute handlebar a_{zH} seat a_{zS}, and crank a_{zC} accelerations, as well as gyro pitch $\ddot{\beta}_y$ accelerations. Due to the sensitivity of these data to numerical noise and sample rates, a 10 Hz low pass filter was used to smooth out transient accelerations. Based on the estimated rider mass, the observed accelerations are utilized to determine the applied inertia loads. This is a very cautious technique, since the IMU is permanently linked to the bike frame, yet the rider's body is greatly damped and functions as a low pass filter for the transmitted bike accelerations.



Terje Rølvåg – Ride

2:33 PM on Monday, December 20, 2021 · Trondheim, Trøndelag

First Ride

Up and down Bromstadekra

0.67 km Distance · 2:11 Moving Time · 7 m Elevation

181 W Weighted Avg Power · 21 kJ Total Work

	Avg	Max
Speed	18.5km/h	30.7km/h
Cadence	70	97
Power	160W	444W
Calories	18	
Temperature	8°C	
Elapsed Time	2:11	

Wahoo ELEMNT BOLT

Add private notes

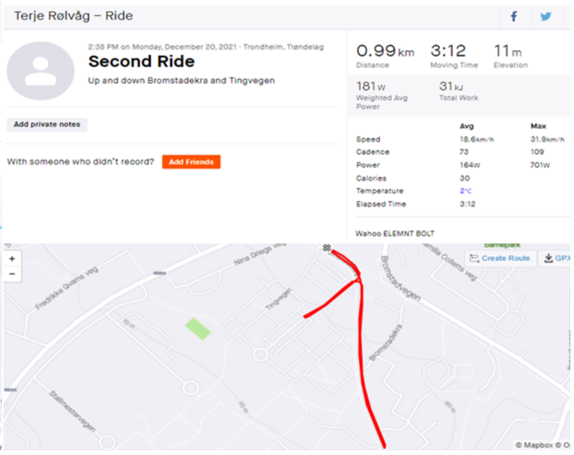
With someone who didn't record? Add Friends

Analyze Your Performance Over Time

Subscribe to automatically match activities on the same route, so you can see how your performance changes over time.

Subscribe

(a) Trip 1



Terje Rølvåg – Ride

2:38 PM on Monday, December 20, 2021 · Trondheim, Trøndelag

Second Ride

Up and down Bromstadekra and Tingvegen

0.99 km Distance · 3:12 Moving Time · 11 m Elevation

181 W Weighted Avg Power · 31 kJ Total Work

	Avg	Max
Speed	18.6km/h	31.9km/h
Cadence	73	109
Power	164W	701W
Calories	30	
Temperature	2°C	
Elapsed Time	3:12	

Wahoo ELEMNT BOLT

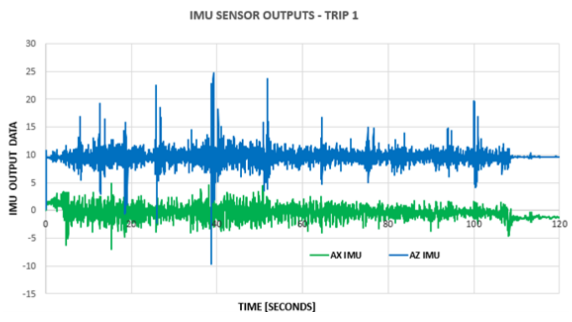
Add private notes

With someone who didn't record? Add Friends

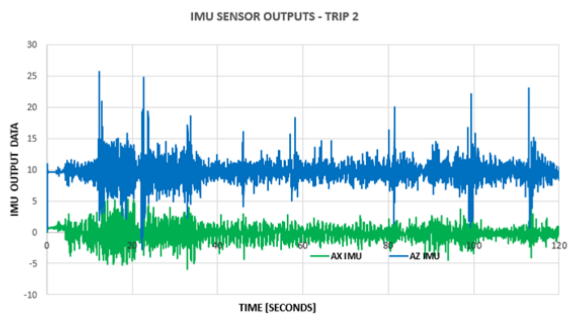
(b) Trip 2

Fig. 6.6. The selected test rides

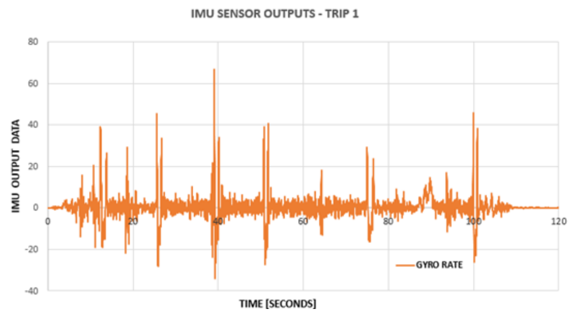
In future implementations, the inverse method will be phased out in favor of force transducers mounted directly to the handlebar and seat tubes to directly sense applied inertia loads. Force-driven digital twins are far more resilient than response-driven digital twins that are impacted by numerical noise generated by accelerometers or strain gauges [14].



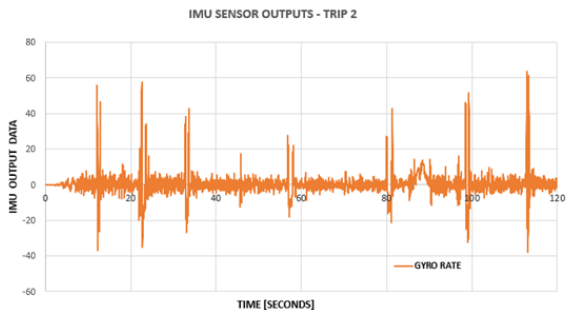
(a) a_x and a_z trip 1



(b) a_x and a_z trip 2



(c) Gyro Rate Trip 1



(d) Gyro Rate Trip 2

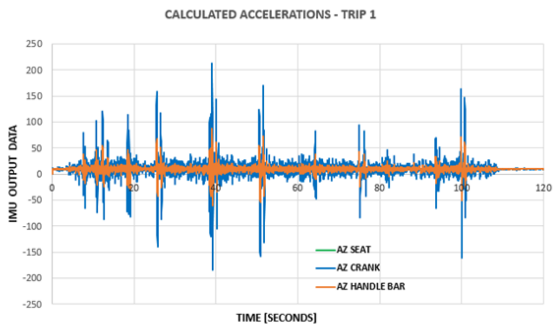
Fig. 6.7. IMU Outputs, trip 1 and trip 2

To determine the unsprung riders body mass that results in inertia loads, a mannequin that properly replicates the rider body's damping and flexibility when aroused by the rigid body bicycle motion is required. This is a difficult problem that will be handled in subsequent effort. Thus, the measured effective unsprung handlebar, seat, and crank rider mass is compared to the physical strain gauge data in a simple test crossing the speed bump depicted in Figure 6.5.

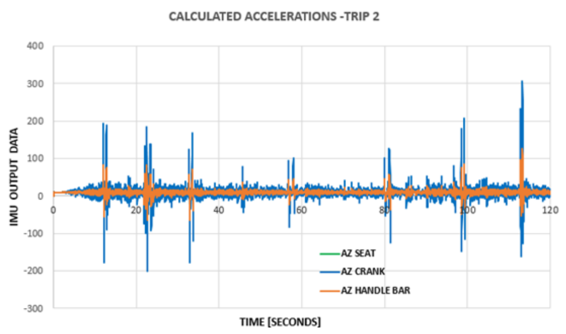
The rigid body handlebar (a_{zH}), seat (a_{zH}), and crank (a_{zH}) accelerations are determined using Eqs. 4.1, 4.2, and 4.3. Figure 6.9 illustrates them. The inverse method uses these estimated accelerations to determine Loads 1-6 in Figure 4.3.

After that, the stresses for the eight virtual gauge's are determined by multiplying the load vector (Load 1-6) by the precomputed FEM-based ROM presented in Table 1. The ROM is confirmed physically, as seen in Figure 6.4. Figure 6.10 and 6.11 illustrates the virtual and actual strain gauge stresses.

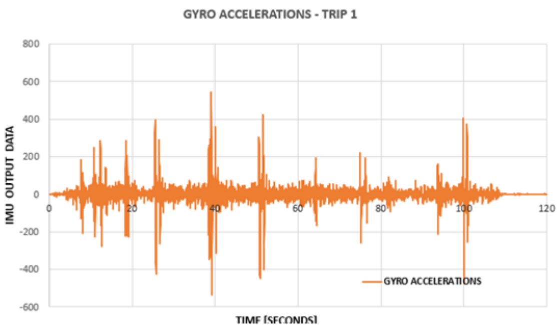
As can be seen from the graphs, the stress correlation is excellent for gauge 1 and 2 positioned on the crank tubes. The stresses in virtual gauges 3, 4, and 8 were too cautious in comparison to the real gauges' results. This is a direct result of the IMU accelerations shown in Figure 6.9, which indicate the rigid body bicycle motion, rather than the sprung / damped acceleration of the rider's body. In future development, a second IMU will be mounted on the rider's body to record the inertia loads delivered to the bicycle frame.



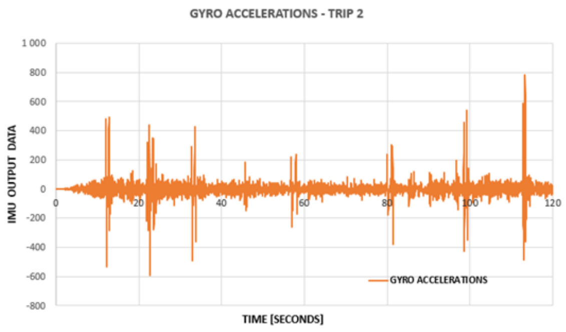
(a) Calculated Accelerations Trip 1



(b) Calculated Accelerations Trip 2

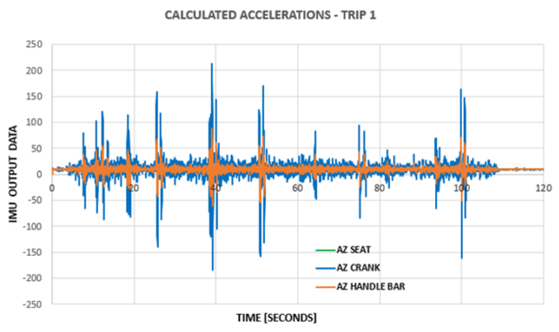


(c) Gyro Accelerations Trip 1

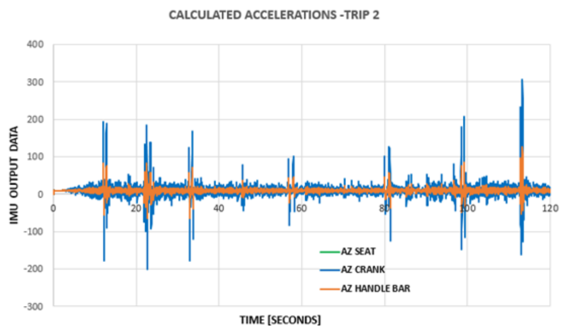


(d) Gyro Accelerations Trip 2

Fig. 6.8. Calculated rigid body accelerations



(a) Calculated forces Trip 1

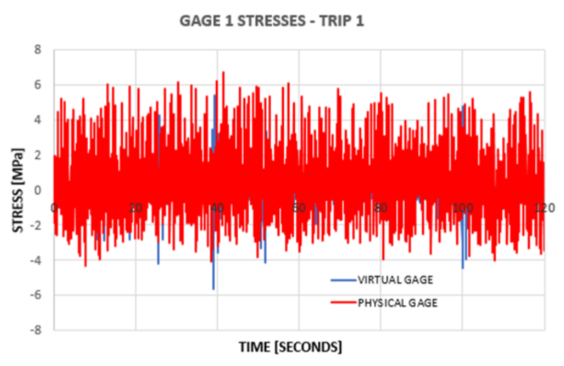


(b) Calculated Forces Trip 2

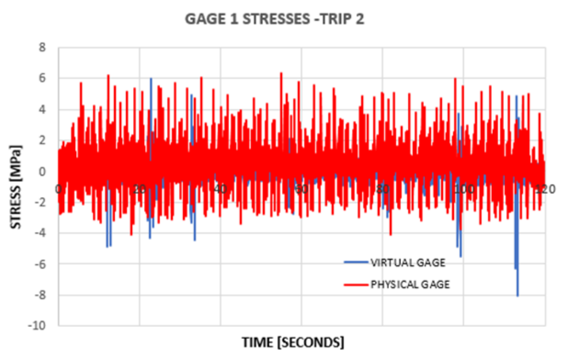
Fig. 6.9. Calculated Ride Loads

Additionally, this will establish the transfer function or dynamic amplification factor for unsprung bike to sprung/damped rider body accelerations. By calibrating the recorded IMU accelerations to achieve a match between virtual and physical strain gauge stresses, the dynamic amplification factors for the handlebar mass, crank mass, and seat mass are calculated to be 0.05-0.1, 0.2-0.4, and 0.5-0.6, respectively.

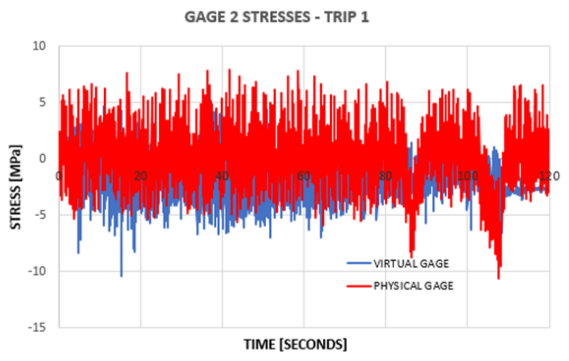
However, this strategy involves prior knowledge of future test results, which is not acceptable. Before submitting a final paper to ICSID2022, physical testing with extra body IMUs will be done to determine the sprung body mass and therefore more accurate inertia forces.



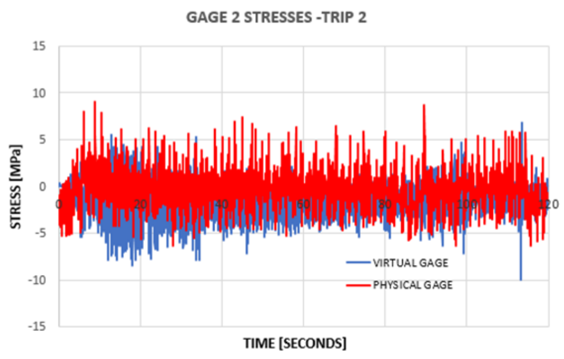
(a) Gauge 1 stresses - trip 1



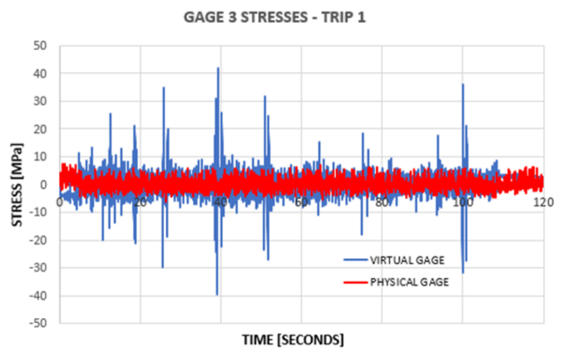
(b) Gauge 1 stresses - trip 2



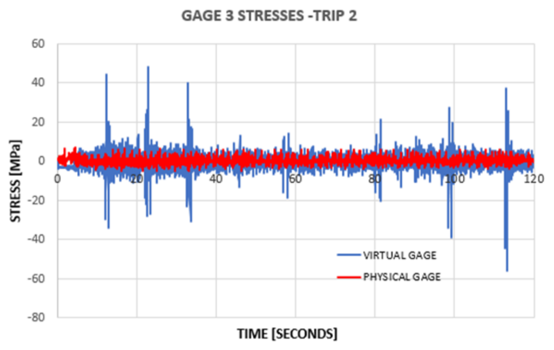
(c) Gauge 2 stresses - trip 1



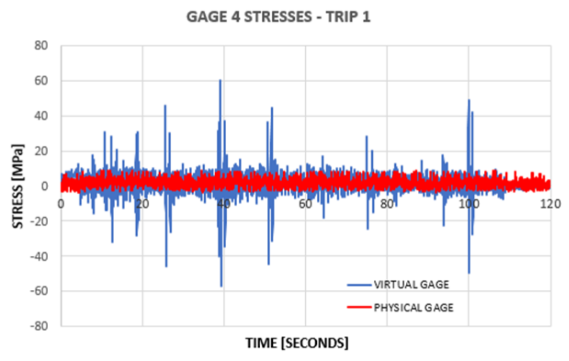
(d) Gauge 2 stresses - trip 2



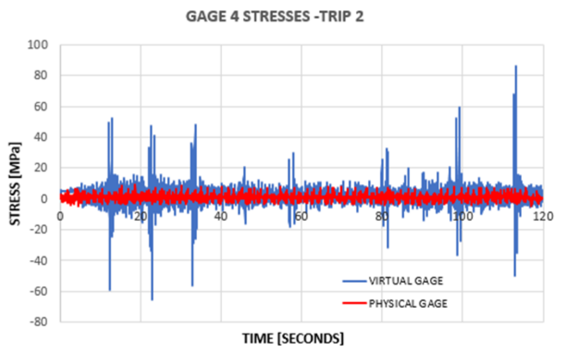
(e) Gauge 3 stresses - trip 1



(f) Gauge 3 stresses - trip 2

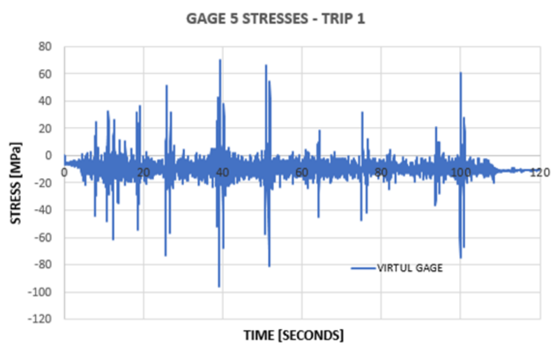


(g) Gauge 4 stresses - trip 1

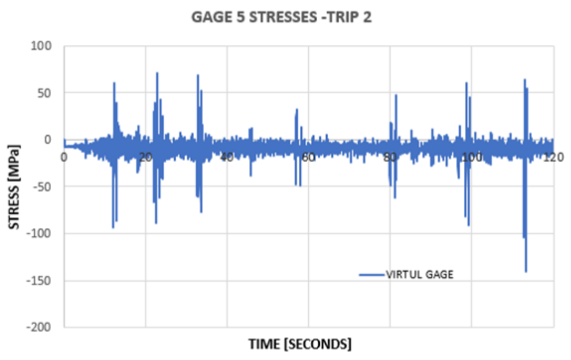


(h) Gauge 4 stresses - trip 2

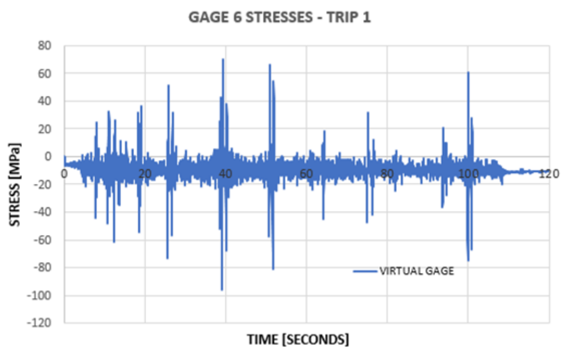
Fig. 6.10. Virtual and physical gauge stresses 1-4 for trip 1 and 2



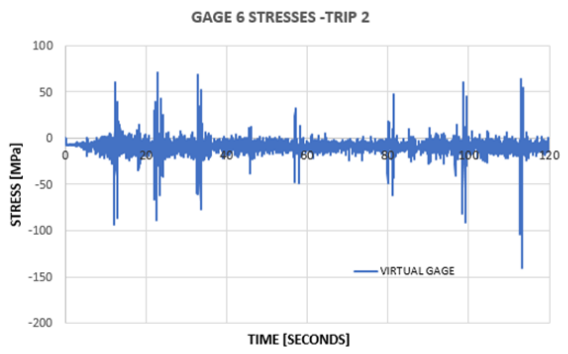
(a) Gauge 5 stresses - trip 1



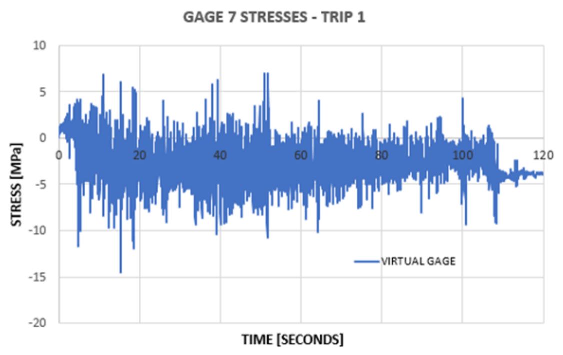
(b) Gauge 5 stresses - trip 2



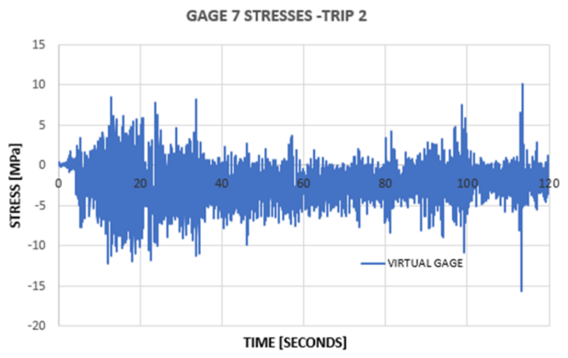
(c) Gauge 6 stresses - trip 1



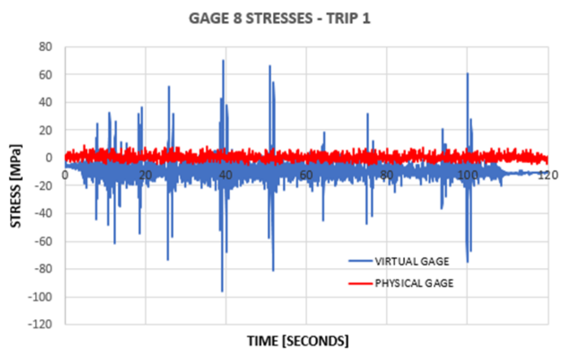
(d) Gauge 6 stresses - trip 2



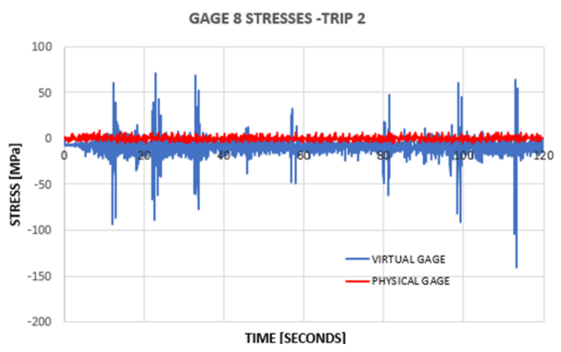
(e) Gauge 7 stresses - trip 1



(f) Gauge 7 stresses - trip 2



(g) Gauge 8 stresses - trip 1



(h) Gauge 8 stresses - trip 2

Fig. 6.11. Virtual and physical gauge stresses 5-8 for trip 1 and 2

Chapter 7

Discussion

7.1 Performance enhancements and encountered bottlenecks

The SHM system built as part of RQ2 is based on a free Mendix application plan that runs on Amazon Web Services and Cloud Foundry. The mendix application enables the customer to access a visual layer through a smartphone in order to gather data published by an assets device, such as the Arduino microcontroller employed in this solution. Thus, the smartphone becomes a critical component in connecting the edge to the cloud. Previously, the inverse method and ROM were coded directly into the microcontroller, but are now incorporated into the cloud. The bicycle is now equipped with six strain gauge sensors and two distinct IMUs and is capable of publishing data at a rate of 100 Hz when monitoring a single characteristic, 74 Hz when monitoring two values, and 60 Hz when monitoring three values. While performance has improved significantly in comparison to last spring's results, it remains deficient as more variables are monitored. Changing from a single value to two values decreases the system's performance by 25%. A 25% decrease is insufficient since the objective was to acquire raw data in real time with a 100 hertz sampling frequency, upload it to the cloud, create a context, and use it for business decisions such as deciding when to do maintenance. These results emphasize the disadvantages of the JavaScript code operating on a single thread, as well as the Bluetooth protocol's inability to reliably transfer messages. According to the author, since JavaScript operates in a single thread and is responsible for parsing data strings and sending messages to the server, a greater amount of messages will be ignored due to the thread being busy. As a result, data will be lost as a direct consequence of this condition, and exploring other methods of delivering data to the server is strongly advised. For example, by uploading directly to the cloud rather than going via the phone using the Bluetooth protocol. Which is possible by using a different application protocol than RESTful HTTP, for instance MQTT.

Due to the smartphone's essential position in this configuration, the system's performance became significantly dependent on the phone's compatibility with the Bluetooth-enabled microcontroller. As previously stated, this implies that additional steps are required to upload data to the cloud, which is now accomplished by connecting the phone to the internet through a 4G connection. The phone, which executes JavaScript code in the browser, becomes a bottleneck for the whole system, raising the issue of why Mendix and Bluetooth were chosen in the first place. If more research on IoT protocols had been conducted during the earlier phases, the architecture of the system may look substantially different than it does now. The system is said to be adaptable enough to switch to a new protocol fairly simply, highlighting once again the system's adaptability. This IoT system is constructed with the use of a very common protocol, the RESTful HTTP.

Bluetooth is a highly coupled technology that does not scale particularly well. This makes the protocol seem out of place in an IoT environment where the aim is reliable data transmission at high rates. However, Bluetooth retains usefulness in the existing system, but not in the function it now serves. The results reveal that although Bluetooth is capable of high-speed communication, the protocol is confined to local usage, implying that IoT application protocols using TCP/DP can overtake its role and publish directly to the cloud.

If the phone's primary function of data transmission to the cloud is delegated to the microcontroller, and the microcontroller instead runs an IoT protocol, the system may become much more Plug-and-Play (PnP). The MQTT protocol, like Bluetooth, is based on the publish and subscribe (pub-sub) pattern and hence provides a lot of advantages and may be capable of overcoming many of the aforementioned difficulties. As a result, MQTT is a promising technology for future systems. Additionally, the Rev2 is compatible with the MQTT protocol and may be used for testing.

Arduino provides a wide variety of microcontrollers, and the one selected for this project was the Arduino Uno WiFi Rev2 (Rev2). One of the factors that contributed to the Rev2's selection was its 5V operating voltage. The Rev2 performed somewhat slower than expected during benchmarking. This fall, the decision was taken to replace the Arduino MKR WiFi 1010 (MKR) with the Arduino Uno WiFi Rev2. The MKR featured a faster CPU but required a voltage converter to operate due to its 3.3V working voltage. As a result, the decision was made to continue with the Rev2. Two similar experiments (not included in this thesis) comparing the speed of these two controllers revealed that the Rev2's loop method took around 10ms, but the MKR's required just under 2ms, indicating that the Rev2's performance had reached its limit in this system. This suggests that while constructing a system with the purpose of sampling sensors at 100 hz or more, a suitably fast CPU must be considered.

7.2 Improvements in modifiability, cloud DT techniques, and Bluetooth problems

The system's strengths remain dependant upon its modifiability. Adding new features is an easy process as long as the developer invests some time in mastering the Mendix platform and Mendix Studio Pro. With the new solution's cloud-based integration of digital twin methods, it becomes easier to reuse methods or algorithms, develop new ones, as well as utilize the multi threaded functionality supported in Mendix. Another advantage is that raw data is now accessible and can be downloaded and displayed from the cloud, while before, data uploaded to the cloud was already processed. This culminates in a more decoupled and thus modifiable system, because consumers using methods or algorithms unrelated to the DT methods can now consume the raw data and produce completely new information. Moreover, a newly developed ROM can be saved in the cloud and utilized to compute new virtual stresses for comparison to older techniques, as well as actual strain gauge data. Another significant advantage of shifting DT techniques to the cloud is the elimination of the necessity for the system to include the Bluetooth protocol. Because the DT methods only rely on raw data which is stored in the cloud, it is feasible to change the role of the phone to only be used as a viewer. In this scenario, the microcontroller running the highly coupled Bluetooth code that connects itself to the phone, can be removed. The notion of removing the Bluetooth protocol, as previously said, will almost certainly resolve the problem of data loss, given that the new communication protocol support larger payloads in order to upload data in batches. Especially when considering that there are different Bluetooth versions around which support maximum payloads of 20 bytes. However, the author wishes to highlight that there are ways to tackle this issue using the current data flow by using Bluetooth versions supporting higher payloads. Anyhow, if the system were to remove the Bluetooth protocol altogether, there would simply be one point of connection (the microcontroller), thus resulting in a far more streamlined process for making the system upload raw data to the cloud by removing unnecessary steps.

The system continues to need user understanding of Bluetooth technology in order to function, which is probably one of the system's primary shortcomings at the moment. To utilize the system, the user must be acquainted of the device's service and characteristic UUIDs. Additionally, certain Bluetooth devices, such as the Thingy52, have various UUIDs associated with their services and features, necessitating the study of their documentation in order to configure the asset using the Thingy52. Thus, the Bluetooth standard becomes a constraint, since unique values must be introduced into the system at some point. Due to the aforementioned issues, the system's server must be somewhat mindful of each and every edge node, thus limiting the system's potential to scale up because the server must be modified to handle a new type of microcontroller. The author contends that a better way would be to make it plug-and-play (PnP), which would imply that once a device is connected and publishes to the cloud, the asset appears magically as a node inside the IoT ecosystem. Because the process of uploading data to the cloud already involves parsing data to and from strings, as demonstrated in the java code in B.2, there is a compelling case to be made for protocols that include these approaches into their architecture. As mentioned in section 2.2, MQTT is one of these technologies which are edge driven, and secure, by only having an outbound port open. Considering that MQTT performs better than RESTful HTTP as well, there is a clear argument for shifting towards a technology such as MQTT if the system wishes to lean more towards IIoT, which can be a more natural approach since the goal is to monitor the health of structures.

7.3 Mendix's capabilities and limitations

One of the alleged benefits of Mendix PaaS is the ability to rapidly design contemporary user interfaces. As the application became more sophisticated, the platform demonstrated its robustness, flexibility, scalability, and speed. There were no indications of server-side bottlenecks throughout the tests made during the thesis, which is due to the restrictions implemented by uploading data to the cloud every 250ms. When it comes to structural health monitoring, one of the thesis's objectives was to construct a dashboard and provide real-time data to the user. Utilizing Mendix for this purpose found to be remarkably difficult, since the Mendix is not pre-configured to produce such dashboards. This meant that throughout development, trade-offs had to be made between adding capabilities to the to allow such dashboards and improving the system's performance. Due to the priority placed on the latter, the UI built this fall is relatively similar to the one developed this spring, with a few notable differences. There are fewer pages, which means that navigating the page requires fewer steps. Additionally, several small improvements have been made to the visuals.

Aside from the limitations of creating the dashboard, one final concern regarding local VS live testing of the Mendix application must be addressed. The capability for generating new users functioned while using a Mendix server hosted locally, but not when using the cloud server. While Mendix has various advantages and disadvantages, this is the one irritation directly tied to testing locally vs in the cloud.

7.4 The difficulties of the inverse method

An inverse method was devised to evaluate the biker-induced inertia loads occurring on the rider throughout the trip. The inverse method's objective is to forecast the distributed dynamic loads delivered to the digital twin model in order to compute the distribution of stresses and strains at the eight hot spots previously selected. While the single IMU accurately captures bicycle movements, the sprung motion of the biker's body is difficult to confirm. Without load cells on the frame tubes, the initial problem was to determine the biker's mass distribution on the handlebars, seat, and crank. The following task was to determine the transfer function or dynamic amplification factor between unsprung bike and sprung/damped cyclist body accelerations. These two points will be addressed in order to improve the estimation of applied inertia loads. The virtual strain gauge stresses will be more accurate since they are a linear function of the anticipated inertia loads. The ultimate objective is to remove the inverse method used by force transducers, resulting in a more robust force-driven digital twin with improved performance. Due to the need for a 100 Hz sampling frequency, simultaneous finite element analysis of a complete finite element model was not possible; rather, the unit-load approach was used on a FE model to locate hotspots using a virtual brittle lacquer methodology in FEDEM. Then, a physical test was used to construct, verify, and approve a ROM impact matrix mapping unit loads to output stresses in the identified hot-spots. This ROM's digital twin solution runs quickly on Arduino, is straightforward to build in Mendix, and is entirely relevant to the linear behavior of a rigid bicycle frame.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

The research questions addressed in the article were answered by building on the thoughts and ideas developed in this spring's project thesis, which served as a foundation for the thesis. To have a better understanding of IoT, IIoT, IoT Framework, IoT Platforms, IoT Ecosystem, and IoT protocols, a systematic review of the literature was undertaken. It was via the use of RQs that additional information on the strengths and limitations of the Mendix PaaS's RESTful HTTP protocol, which was a vital component of the IoT architecture, was made available. Findings from testing reveal that the system can operate at 100 Hz while monitoring only one Bluetooth characteristic. This indicates that the system is capable of connecting devices to an IoT ecosystem and addressing issues, qualifying it as an IoT platform and thus deserving of the term ”Internet of Things”. In addition, techniques for calibrating strain gauges and IMU's are presented. While working on this thesis, a generic IoT framework was created that can also be easily developed and maintained in the future by someone with less programming experience. The adoption of well-established software development processes allows for the successful implementation of strategies. Anyone who is interested in contributing to the development of the system will find documentation to be of assistance. This is possible since the author employed the software lifecycle as one of his ways to solve the system's difficulties. This is a realistic expectation. Because of the system's high adaptability, flexibility, and interoperability, there are several chances to branch out and develop higher-performing nodes within the ecosystem, which speaks well for the system's long-term viability.

The inverse method was utilized to calculate the most essential inertia loads that were experienced during bicycle rides in the mountains. The technique, on the other hand, revealed substantial limits as a consequence of the low-cost equipment that was used. Rider body accelerations recorded by the IMU are too cautious and do not adequately represent the damped rider body accelerations. Consequently, the inertia loads and gauge stresses that have been estimated are a little too conservative. A mannequin model or additional accelerometers mounted on the rider may be utilized to better forecast the sprung motion and hence the cyclist inertia loads, but both options increase complexity to the system and so reduce its bandwidth and performance.

A ROM was shown, and it was demonstrated how it could be utilized to correctly represent the whole bike frame FE model while also giving gauge stress calculation. The idea of future enhancements to the ROM is intriguing to consider.

The use of a digital twin to power SHM is still in its infancy. The authors looked at a variety of IoT systems that claimed to be capable of delivering the speed and flexibility required for real-time SHM. The vast majority of IoT technologies are designed to aid with logistics, rather than to offer the high-speed edge solutions necessary for real-time SHM operations. Accordingly, the authors plan to develop a customized SHM solution that is not Bluetooth-enabled and is based on MQTT while keeping the Mendix app for display of edge data that has been transferred to a cloud service provider in the background.

8.2 Future Work

Despite the fact that the solution represents a substantial advance, there is still room for development, both in terms of the IT system and in terms of the DT approaches. Areas for improvement are proposed in the following areas: (1) Getting rid of the smartphone's role as an intermediary for the microcontroller and the cloud, and instead using it as a basic data reader for online or offline data. (2) Examine the viability of employing existing IoT protocols, such as MQTT, for data upload to the cloud, therefore making the system PnP, faster, and more secure. (3) Reduce the number of redundant operations and integration between system nodes. (4) Create a dashboard using Mendix or an equivalent piece of software for showing the data that has been provided. (5) To increase the precision of the results, a mannequin model or extra accelerometers installed on the rider can be used. (6) Increase the size of the ROM in order to account for the impact of bicycle off-road handling loads.

References

[1] Denyer Smart Tranfield. ‘Towards a methodology for developing evidence - informed management knowledge by means of systematic review’. In: *British Journal of Management, Vol. 14 No. 3* (2003), pp. 207–222.

[2] Kitchenham Charters. ‘Guidelines for performing systematic literature reviews in software engineering’. In: *Technical Report EBSE-2007-01, School of Computer Science and Mathematics, Keele University* (2007).

[3] Mujtaba Mattsson Petersen Feldt. ‘Systematic Mapping Studies in Software Engineering’. In: *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering. 17* (2008), p. 17.

[4] L. Bass, P. Clements and R. Kazman. *Software Architecture in Practice - Third Edition*. Addison Wesley, 2012.

[5] I. Grigorik. ‘Making the web faster with http 2.0’. In: *Communications of the ACM, vol. 56, no. 12* (2013), pp. 42–49.

[6] N. S. Han. ‘Semantic service provisioning for 6lowpan: powering internet of things applications on web’. In: *Ph.D. dissertation, Institut National des Telecommunications* (2015), p. 4.

[7] Covill et al. ‘An Assessment of Bicycle Frame Behaviour under Various Load Conditions Using Numerical Simulations, 11th conference of the International Sports Engineering Association’. In: *ELSEVIER Procedia Engineering* (2016), pp. 665–670.

[8] Alasdair Gilchrist. *Industry 4.0, Industrial Internet of Things*. Apress, 2016.

[9] D Borycki. *Programming for the Internet of Things: Using Windows 10 IoT Core and Azure IoT Suite*. Microsoft Press, 2017.

[10] Pallavi Sethi and Smruti R. Sarangi. ‘Internet of Things: Architectures, Protocols, and Applications’. In: *Journal of Electrical and Computer Engineering, Volume 2017* (2017), p. 25.

[11] M. Bella T. Rølvåg. ‘Dynamic test bench for motocross engines’. In: *Adv. Mech. Eng. 9* (2017).

[12] Sakina Elhadi et al. ‘Comparative Study of IoT Protocols’. In: *Smart Application and Data Analysis for Smart Cities (SADASC’18)* (2018), p. 5.

[13] T. Rølvåg et al. ‘Fatigue analysis of high performance race engines’. In: *3rd International Conference on Structural Integrity and Durability* (2019).

[14] T. Rølvåg T. Moi A. Cibicik. ‘Digital twin based fatigue monitoring of a knuckle boom crane’. In: *published at 3rd International Conference on Structural Integrity and Durability* (2019).

[15] Seleznev Yakovlev. ‘Industrial Application Architecture IoT and protocols AMQP, MQTT, JMS, REST, CoAP, XMPP, DDS’. In: *International Journal of Open Information Technologies, 7(5)* (2019), pp. 17–28.

[16] Sarbani Roy ByJoy Dutta. *Universal IoT Framework*. CRC Press, 2020.

[17] Puneet Mathur. *IoT Machine Learning Applications in Telecom, Energy, and Agriculture*. Apress, 2020.

[18] Abdallah Alami Sidna Amine. ‘Analysis and evaluation of communication protocols for IoT applications’. In: *in Proceedings of the 13th International Conference on Intelligent Systems: Theories and Applications* (2020), pp. 1–6.

[19] Hui Li Yuequan Bao. ‘Machine learning paradigm for structural health monitoring’. In: *Sage* (2020), p. 5.

[20] Uwiringiyimana et al. ‘IoT Platforms, Use Cases, Privacy, and Businesss Models’. In: *Springer Nature Switzerland AG* (2021), pp. 21–49.

[21] U. Sinthuja Dr. S. Thavamani. ‘MQTT Messages-An Overview’. In: *IJMCR Volume 09* (2021), p. 4.

[22] Bilal Şenol Muhammed Yıldırım Uğur Demiroğlu. ‘An in depth exam of IoT, IoT Core Components, IoT Layers, and Attack Types’. In: *European Journal of Science and Technology (28)* (2021), pp. 665–669.

[23] Petter S Rolvåg. ‘Structural Health Monitoring Project Thesis’. In: *Structures and Their Analysis, Springer* (2021), pp. 387–411.

[24] Amazon. *What is AWS?* URL: https://aws.amazon.com/what-is-aws/?nc1=f_cc.

[25] Arduino. *Arduino 9-Axis Motion Shield*. URL: https://store-usa.arduino.cc/products/arduino-9-axis-motion-shield (visited on 11/06/2021).

[26] Elecrow. *Crowtail Strain Gauge Sensor Module*. URL: https://www.elecrow.com/crowtail-base-shield-p-1264.html (visited on 11/06/2021).

[27] Elecrow. *Crowtail-Base-Shield for Arduino 2.0*. URL: https://www.elecrow.com/crowtail-base-shield-p-1264.html (visited on 11/06/2021).

[28] Cloud Foundry. *Cloud Foundry*. URL: https://docs.cloudfoundry.org/.

[29] Cloud Foundry. *Cloud Foundry Members*. URL: https://www.cloudfoundry.org/members/ (visited on 11/06/2021).

[30] HBM. *catman Data Acquisition Software: Connect. Measure. Visualize. Analyze.* URL: https://www.hbm.com/en/2290/catman-data-acquisition-software/ (visited on 11/06/2021).

[31] HBM. *Spider8 Measurement Technology.* URL: https://usas.no/index.php?page=shop.getfile&file_id=26&product_id=38&option=com_virtuemart&Itemid=2 (visited on 11/06/2021).

[32] Tim Hunter. *TrueLoads Trek Bicycles Takes Simulation to the Extreme.* URL: https://www.wolfstartech.com/trek (visited on 05/01/2022).

[33] IBM. *Application Programming Interface.* URL: https://www.ibm.com/topics/api.

[34] Mendix. *Mendix Academy.* URL: https://academy.mendix.com/link/paths (visited on 11/04/2021).

[35] SM. *ArduinoBLE.* URL: https://www.arduino.cc/en/Reference/ArduinoBLE (visited on 11/04/2021).

[36] TML. *TML Strain Gauge.* URL: https://tml.jp/e/product/strain_gauge/f_list.html (visited on 11/06/2021).

List of Figures

List of Tables

Appendix A

Javascript Code

A.1 BLECharacteristicSubscribe.js

```javascript
// This file was generated by Mendix Studio Pro.
//
// WARNING: Only the following code will be retained when actions are
    regenerated:
// - the import list
// - the code between BEGIN USER CODE and END USER CODE
// - the code between BEGIN EXTRA CODE and END EXTRA CODE
// Other code you write will be lost the next time you deploy the project.
import { Big } from "big.js";

// BEGIN EXTRA CODE

/**
 * The functions below are required for reading specific datatypes. Each
    function should have the same params and return the same object.
 * Reads incoming data from an event and appends it to the datastring.
 * @param Event event
 * @param String datastring
 * @returns String dataString
 */

function GaugeData_Measured(event, ds) {
  ds += (event.target.value.getUint32(0, true)/1000).toFixed(2) + ",";
  ds += event.target.value.getInt16(4, true)/100 + ",";
  ds += event.target.value.getInt16(6, true)/100 + ",";
  ds += event.target.value.getInt16(8, true) + ",";
  ds += event.target.value.getInt16(10, true) + ",";
  ds += event.target.value.getInt16(12, true) + ",";
  ds += event.target.value.getInt16(14, true) + ",";
  ds += event.target.value.getInt16(16, true) + ",";
  ds += event.target.value.getInt16(18, true) + ";";
  return ds;
}
```

```

function uint(event, ds) {
  ds += event.target.value.getUint16(0, true) + ";";
  return ds
}
// END EXTRA CODE

/**
 * Subscribes to a characteristic given a UUID and commits data to the server on every notification retrieved from the peripheral device.
 * @param {string} serviceUUID
 * @param {string} characteristicUUID
 * @param {MxObject} eventReference
 * @param {MxObject} eventBatch
 * @param {string} dataType
 * @returns {Promise.<string>}
 */
export async function BLECharacteristicSubscribe(serviceUUID, characteristicUUID, eventReference, eventBatch, dataType) {
  // BEGIN USER CODE

  var gattServer = window.gattServer;

  let startTime = Date.now();
  let dataString = new String(eventReference.getGuid() +":");

  if(gattServer){
    return new Promise((resolve, reject) => {
      gattServer.getPrimaryService(serviceUUID)
      .then( (s) => {
        return s.getCharacteristic( characteristicUUID );
      })
      .then(characteristic => {
        characteristic.startNotifications()
        .then(() => {
          eventBatch.set("IsLogging", true);
        })
        .then(() => {
          characteristic.addEventListener(
              'characteristicvaluechanged',
            (event) => {

              switch (dataType) {
                // Switch statement to decide which function to run to parse the bluetooth data.
                case "GaugeData_Measured": dataString = GaugeData_Measured(event, dataString); break;
                case "uint": dataString = uint(event, dataString); break;
              }
              var elapsedTime = Date.now() - startTime;

              if(elapsedTime > 250){
                startTime = Date.now();
                let newDataString = new String(dataString);

                mx.data.create({
                  entity: "SHM_Module.Data",
                  callback: function(obj) {
                    obj.set('data', newDataString);
                    mx.data.action({
                      params: {
                        applyto: "selection",
                        actionname: "SHM_Module.SendDataToServer",
                        guids: [obj.getGuid()],
                        async: false
                      },
                      callback: function(obj) {
                        resolve(obj);
                      },
                      error: function(error) {
                        alert(error.message);
                      },
                      onValidation: function(validations) {
                        alert("There were " + validation.length + " validation errors");
                      }
                    });
                  },
                  error: function(e) {
                    console.error("Could not commit object:", e);
                  }
                });
                dataString = new String(eventReference.getGuid() +":");
              }
            }
          );
          resolve(true);
        })
        .catch(error => console.error(error.code, error.name, error.message))
      });
    });
  } else{
    return Promise.reject("No gatt server found");
  }
  // END USER CODE
}
```

A.2 BLEDeviceConnect.js

```
// This file was generated by Mendix Studio Pro.
//
// WARNING: Only the following code will be retained when actions are
    regenerated:
// - the import list
// - the code between BEGIN USER CODE and END USER CODE
// - the code between BEGIN EXTRA CODE and END EXTRA CODE
// Other code you write will be lost the next time you deploy the project.
import { Big } from "big.js";

// BEGIN EXTRA CODE
// END EXTRA CODE

/**
 * Pairs with a Bluetooth device with a given UUID.
 * @param {string} primaryServiceUUID
 * @param {MxObject} eventBatch
 * @returns {Promise.<void>}
 */
export async function BLEDeviceConnect(primaryServiceUUID, eventBatch) {
  // BEGIN USER CODE
  var bluetoothDevice = window.bluetoothDevice;

  try{
    if(!navigator.bluetooth){
      alert("This device does not support bluetooth.")
      return;
    }
    let encoder = new TextEncoder('utf-8');

    return new Promise((resolve, reject) => {
      var gattServer = window.gattServer;
      if(gattServer && gattServer.connected){

        resolve(true);
      } else{
        navigator.bluetooth.requestDevice({
          filters: [{ services: [primaryServiceUUID] }]
          //optionalServices: [optionalServiceUUID]
          })
          .then((device) => {
            return device.gatt.connect()
          })
          .then((server) => {
            console.log(server);
            window.gattServer = server;
            eventBatch.set("IsDeviceConnected", true);
            console.log("Connected to new server");
            resolve(true);
          });
      }
    })
  } catch (error) {
    console.log(error);
    return Promise.resolve(false);
  }
  // END USER CODE
}
```

A.3 BLEDeviceDisconnect.js

```
// This file was generated by Mendix Studio Pro.
//
// WARNING: Only the following code will be retained when actions are
    regenerated:
// - the import list
// - the code between BEGIN USER CODE and END USER CODE
// - the code between BEGIN EXTRA CODE and END EXTRA CODE
// Other code you write will be lost the next time you deploy the project.
import { Big } from "big.js";
// BEGIN EXTRA CODE
// END EXTRA CODE

/**
 * Disconnect from all Bluetooth devices.
 * @param {string} serviceUUID
 * @param {string} characteristicUUID
 * @param {MxObject} eventBatch
 * @returns {Promise.<string>}
 */
export async function BLEDeviceDisconnect(serviceUUID, characteristicUUID,
    eventBatch){
  // BEGIN USER CODE
  var gattServer = window.gattServer;
  if (gattServer) {
    gattServer.disconnect();
    console.log('Bluetooth Device is disconnected');
    eventBatch.set("IsDeviceConnected", false);
    eventBatch.set("IsLogging", false);
      return;
  }
  else {
    console.log('Bluetooth Device is already disconnected');
  }
  // END USER CODE
}
```

Appendix B

Java Code

B.1 computeInverseData.java

```java
// This file was generated by Mendix Studio Pro.
//
// WARNING: Only the following code will be retained when actions are regenerated:
// - the import list
// - the code between BEGIN USER CODE and END USER CODE
// - the code between BEGIN EXTRA CODE and END EXTRA CODE
// Other code you write will be lost the next time you deploy the project.
// Special characters, e.g.,    ,    ,    , etc. are supported in comments.

package shm_module.actions;

import java.math.BigDecimal;
import java.util.Iterator;
import com.mendix.core.Core;
import com.mendix.systemwideinterfaces.core.IContext;
import com.mendix.webui.CustomJavaAction;
import com.mendix.systemwideinterfaces.core.IMendixObject;

public class computeInverseData extends CustomJavaAction<java.lang.Void>
{
  private java.util.List<IMendixObject> __acc_gyro_list;
  private java.util.List<shm_module.proxies.acc_gyro> acc_gyro_list;
  private java.math.BigDecimal mass;

  public computeInverseData(IContext context, java.util.List<IMendixObject> acc_gyro_list, java.math.BigDecimal mass)
  {
    super(context);
    this.__acc_gyro_list = acc_gyro_list;
    this.mass = mass;
  }
```

```java

@java.lang.Override
public java.lang.Void executeAction() throws Exception
{
  this.acc_gyro_list = new java.util.ArrayList<shm_module.proxies.acc_gyro>();
  if (__acc_gyro_list != null)
    for (IMendixObject __acc_gyro_listElement : __acc_gyro_list)
      this.acc_gyro_list.add(shm_module.proxies.acc_gyro.initialize(getContext(), __acc_gyro_listElement));

  // BEGIN USER CODE


  // We dont want to do anything unless the list is of size 3 or higher.
  if(__acc_gyro_list.size()<3) {
    return null;
  }

  // torque and radius is not yet supported.
  float torque = 0;
  float radius = 1;


  // We ignore the first and last element of the list.
  for (int i = 1; i < __acc_gyro_list.size()-2; i++) {

    //Timer values for calculating the dt
    java.math.BigDecimal t0 = (BigDecimal) __acc_gyro_list.get(i-1).getMember(getContext(), "Timer");
    java.math.BigDecimal t2 = (BigDecimal) __acc_gyro_list.get(i+1).getMember(getContext(), "Timer");

    // IMU accelerations for the inverse method
    java.math.BigDecimal ax = (BigDecimal) __acc_gyro_list.get(i-1).getMember(getContext(), "acc_x");
    java.math.BigDecimal az = (BigDecimal) __acc_gyro_list.get(i-1).getMember(getContext(), "acc_z");

    // IMU pitch rate for the handlebar, crank, and seat acceleration calculations.
    java.math.BigDecimal dBeta0 = (BigDecimal) __acc_gyro_list.get(i-1).getMember(getContext(), "Timer");
    java.math.BigDecimal dBeta2 = (BigDecimal) __acc_gyro_list.get(i+1).getMember(getContext(), "Timer");

    float dt = t2.floatValue() - t0.floatValue();

    float ddBeta1 = (dBeta2.floatValue() - dBeta0.floatValue()) / (2*dt);

    // Calculating the handlebar, crank and seat accelerations
    float az_H = 0.9f * dBeta0.floatValue() + az.floatValue();
    float az_C = 0.38f * dBeta0.floatValue() + az.floatValue();
    float az_S = 0.15f * dBeta0.floatValue() + az.floatValue();

    // The list containing the final input loads
    float[] inputLoads = {
        mass.floatValue()*ax.floatValue(),//Input load 1
        0.2f*mass.floatValue()*az_H,      //Input load 2
        0.5f*mass.floatValue()*az_S,      //Input load 3
        0,                    //Input load 4 is calculated later..
        0.3f*mass.floatValue()*az_C,      //Input load 5
        torque/radius            //Input load 6
    };

    //Input load 4 is calculated here.
    if(ax.floatValue() < 0) {
      inputLoads[3] = mass.floatValue()*ax.floatValue();
    }
    else {
      inputLoads[3] = 0.02f*mass.floatValue()*az.floatValue();
    }


    float[] outputLoads = {0, 0, 0, 0, 0, 0, 0, 0};

    // "Matrix multiplication" to get the output loads
    for (int k = 0; k < 8; k++) {
          for (int j = 0; j < 6; j++) {
            outputLoads[k] += unitLoadMatrix[k][j]*inputLoads[j]/1000000;
          }
      }

    // Creating inverse data on the server
    IMendixObject inverseData = Core.instantiate(getContext(), "SHM_Module.inverse_data");

    // Setting the inverseData member to the corresponding output value.
    inverseData.setValue(getContext(), "gauge1", BigDecimal.valueOf(outputLoads[0]));
    inverseData.setValue(getContext(), "gauge2", BigDecimal.valueOf(outputLoads[1]));
    inverseData.setValue(getContext(), "gauge3", BigDecimal.valueOf(outputLoads[2]));
    inverseData.setValue(getContext(), "gauge4", BigDecimal.valueOf(outputLoads[3]));
    inverseData.setValue(getContext(), "gauge5", BigDecimal.valueOf(outputLoads[4]));
    inverseData.setValue(getContext(), "gauge8", BigDecimal.valueOf(outputLoads[7]));

    // Setting the acc gyro associativity
    inverseData.setValue(getContext(), "SHM_Module.inverse_data_acc_gyro", __acc_gyro_list.get(i).getId());
```

```java

      // Commiting the changes made to the inverse data.
      Core.commit(getContext(), inverseData);
    }
    return null;
    // END USER CODE
  }

  /**
   * Returns a string representation of this action
   */
  @java.lang.Override
  public java.lang.String toString()
  {
    return "computeInverseData";
  }

  // BEGIN EXTRA CODE
  private static float[][] unitLoadMatrix = {
      {262.002f, -4145.12f, -1245.9f, 9881.25f, -730.562f,  459.998f},
        {10845.7f, 3200.66f,  -2324.07f,  -18718f, -1853.65f,  -3949.82f},
        {-13811.4f, -28905.200f,  -6017.640f, 84544.000f, -8588.380f, 40.441f},
        {15628.700f,  25590.600f, 11752.200f, -78251.400f,  16393.400f, -98.947f},
        {18988.300f,  -8742.240f, -30341.600f,  2405.160f,  -20489.200f, 845.881f},
        {18719.700f,  -8818.380f, -30318.700f,  2856.210f,  -20426.000f, 935.281f},
        {15079.400f,  4344.640f,  -3474.600f, -25773.100f,  -3227.950f, -4762.280f},
        {10230.400f,  2761.440f,  -1975.270f, -16989.900f,  -4828.810f, -226.252f}
        };
  // END EXTRA CODE
}
```

B.2 dataParseCommit.java

```java
// This file was generated by Mendix Studio Pro.
//
// WARNING: Only the following code will be retained when actions are regenerated:
// - the import list
// - the code between BEGIN USER CODE and END USER CODE
// - the code between BEGIN EXTRA CODE and END EXTRA CODE
// Other code you write will be lost the next time you deploy the project.
// Special characters, e.g.,    ,    ,    , etc. are supported in comments.

package shm_module.actions;

import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Set;
import com.mendix.core.Core;
import com.mendix.core.objectmanagement.member.MendixObjectReference;
import com.mendix.systemwideinterfaces.core.IContext;
import com.mendix.systemwideinterfaces.core.IMendixIdentifier;
import com.mendix.systemwideinterfaces.core.IMendixObject;
import com.mendix.systemwideinterfaces.core.IMendixObjectMember;
import com.mendix.systemwideinterfaces.core.meta.IMetaPrimitive;
import com.mendix.webui.CustomJavaAction;
import com.mendix.logging.ILogNode;

public class dataParseCommit extends CustomJavaAction<java.lang.Void>
{
  private java.lang.String dataString;

  public dataParseCommit(IContext context, java.lang.String dataString)
  {
    super(context);
    this.dataString = dataString;
  }

  @java.lang.Override
  public java.lang.Void executeAction() throws Exception
  {
    // BEGIN USER CODE
    if(dataString == null) {
      LOG.info("Payload is null.");
      return null;
    }

    String[] dataStringSplit = dataString.split(":");

    if(dataStringSplit.length == 1) {
      LOG.info("Payload comes with no data.");
      return null;
    }

    IMendixIdentifier eventIdentifier = Core.createMendixIdentifier(dataStringSplit[0]);
    IMendixObject event = Core.retrieveId(getContext(), eventIdentifier);
    String dataTypeString = "SHM_Module.";

    IMendixObjectMember eventCharacteristicMember = event.getMember(getContext(), "SHM_Module.Event_Characteristic");
    IMendixIdentifier characteristicId = (IMendixIdentifier) eventCharacteristicMember.getValue(getContext());
    IMendixObject characteristic = Core.retrieveId(getContext(), characteristicId);
    IMendixObjectMember characteristicDataTypeMember = characteristic.getMember(getContext(), "SHM_Module.Characteristic_DataType");
    IMendixIdentifier dataTypeId = (IMendixIdentifier) characteristicDataTypeMember.getValue(getContext());
    IMendixObject dataType = Core.retrieveId(getContext(), dataTypeId);

    dataTypeString += dataType.getValue(getContext(), "Name");

    String[] data = dataStringSplit[1].split(";");
    IMendixObject objEntry = Core.instantiate(getContext(), dataTypeString);

    Collection<? extends IMetaPrimitive> metaPrimitives = objEntry.getMetaObject().getMetaPrimitives();

    if(data[0].split(",").length != metaPrimitives.size()) {
      LOG.info("Payload data is not compatible with the selected datatype.");
      LOG.info(data[0].split(",").length + " != "  + metaPrimitives.size() + ",\tDataType: " + dataTypeString );
      return null;
    }

    for (int i = 0; i < data.length; i++) {
      String[] entry = data[i].split(",");

      IMendixObject gageDataEntry = Core.instantiate(getContext(), dataTypeString);

      int j = 0;
      for(IMetaPrimitive metaPrimitive : metaPrimitives) {
        gageDataEntry.getMember(getContext(), metaPrimitive.getName()).parseValueFromString(getContext(), entry[j]);
        j++;
      }
      gageDataEntry.setValue(getContext(), "SHM_Module.Data_Event", event.getId()); //dataTypeString+"_Event"

      Core.commit(getContext(), gageDataEntry);
    }
    return null;
    // END USER CODE
  }

  /**
   * Returns a string representation of this action
   */
  @java.lang.Override
  public java.lang.String toString()
  {
    return "dataParseCommit";
  }

  // BEGIN EXTRA CODE

  public static ILogNode LOG = Core.getLogger("RobTest");
  // END EXTRA CODE
}
```

Appendix C

Arduino Code

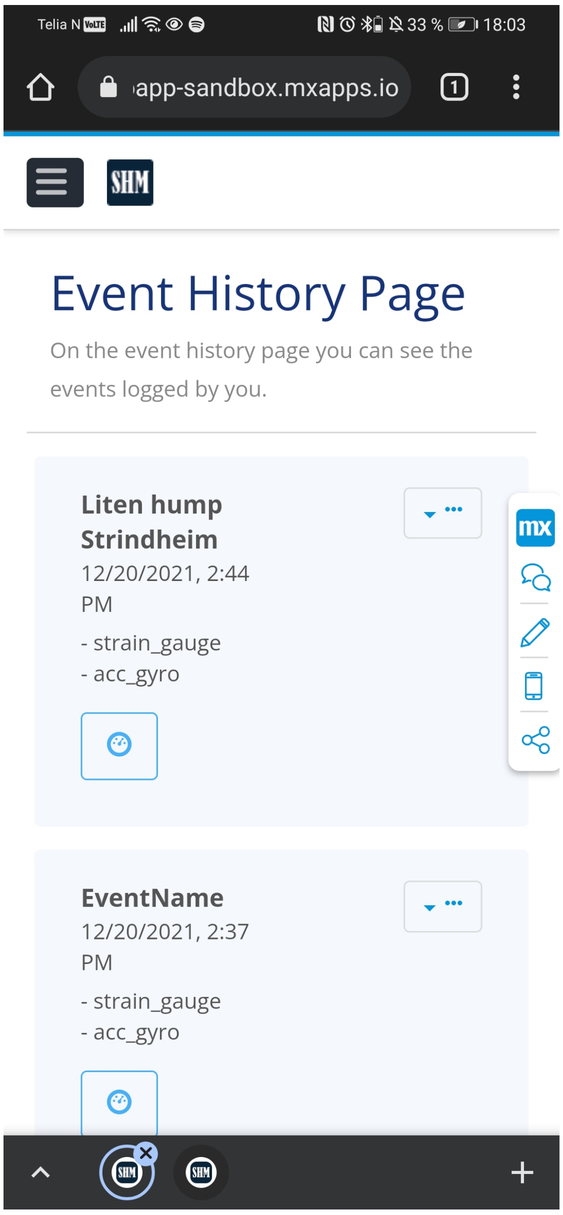
C.1 UNO_Wifi_BLE.ino

```
#include <ArduinoBLE.h>
#include "Arduino_NineAxesMotion.h"
#include <Wire.h>
#include <Arduino_LSM6DS3.h>

/*  This program uses two libraries: ArduinoBLE and Arduino_NineAxesMotion.
    These libraries are important for understanding this program.
    Examples located in the github repos are  great for learnings.
    https://www.arduino.cc/en/Reference/ArduinoBLE,
    https://github.com/arduino-libraries/ArduinoBLE
    https://github.com/arduino-libraries/Arduino_NineAxesMotion
    https://www.arduino.cc/en/Reference/ArduinoLSM6DS3
*/

// Holds the values read from the strain gauge sensor
union strain_gauge {
  struct __attribute__((packed)) {
    unsigned long timer;
    short gauge1;
    short gauge2;
    short gauge3;
    short gauge4;
    short gauge5;
    short gauge8;
  };
  byte bytes[16];
};

// Holds the values read from the IMU sensor
union acc_gyro {
  struct __attribute__((packed)) {
    unsigned long timer;
    short acc_x;
    short acc_z;
    short gyro_y;
  };
  byte bytes[10];
};

union strain_gauge strain_gauge_data; // Stores strain gauge values
union acc_gyro acc_gyro_data;         // Stores IMU values
union acc_gyro grav_gyro_data;        // Stores IMU values

NineAxesMotion IMUSensor; // The IMUSensor object for reading and updating
    the 9-axis motion shield mounted on the Arduino.
BLEService structural_health_data_Service("19b10000-e8f2-537e-4f6c-
    d104768a1214"); // Creating the bluetooth service

// Creating the bluetooth characteristics
BLECharacteristic strain_gauge_data_characteristic("19b10001-e8f2-537e-4f6c
    -d104768a1214", BLERead | BLENotify, sizeof(strain_gauge_data));
BLECharacteristic acc_gyro_data_characteristic("19b10002-e8f2-537e-4f6c-
    d104768a1214", BLERead | BLENotify, sizeof(acc_gyro_data));
BLECharacteristic grav_gyro_data_characteristic("19b10003-e8f2-537e-4f6c-
    d104768a1214", BLERead | BLENotify, sizeof(grav_gyro_data));

//Pins used for the LED
int redLEDPin = 5;
int greenLEDPin = 4;
int blueLEDPin = 3;

// Timer values for the LED blink functionality
long blinkTimer = 0;
long previousMillis = 0;

// Sample hold time in milliseconds. Value of 10 gives 1000/10 = 100 Hz
    sample frequency.
unsigned long sampleAndHoldTime = 10; //

void setup() {
  //Serial.begin(9600);  //Un-comment if you wish to read value from the
    serial monitor.

  // Setting output pin modes for the LEDs.
  pinMode(redLEDPin, OUTPUT);
  pinMode(greenLEDPin, OUTPUT);
  pinMode(blueLEDPin, OUTPUT);

  // Setting input pin modes.
  pinMode(A0, INPUT);
  pinMode(A1, INPUT);
  pinMode(A2, INPUT);
  pinMode(A3, INPUT);
  pinMode(A4, INPUT);
  pinMode(A5, INPUT);

  Wire.begin(); //Initialize I2C communication to the let the library
    communicate with the sensor.
  //Sensor Initialization
  IMUSensor.initSensor(); //The I2C Address can be changed here inside this
     function in the library
  IMUSensor.setOperationMode(OPERATION_MODE_NDOF);  //Can be configured to
    other operation modes as desired
  IMUSensor.setUpdateMode(MANUAL);  //The default is AUTO. Changing to
    manual requires calling the relevant update functions prior to calling
    the read functions
  //Setting to MANUAL requires lesser reads to the sensor
  IMUSensor.updateAccelConfig();

  if (!BLE.begin()) {
    //Serial.println("BLE initialization failed!"); // Un-comment if using
    serial.
    while (1);
  }

  if (!IMU.begin()) {
    //Serial.println("BLE initialization failed!"); // Un-comment if using
    serial.
    while (1);
  }

  //Serial.println("BLE Central - Peripheral Explorer");  Un-comment if
    using serial.
  // Bluetooth Setup
  BLE.setLocalName("SHM - UNO_WIFI");
  BLE.setAdvertisedService(structural_health_data_Service);
  structural_health_data_Service.addCharacteristic(
    acc_gyro_data_characteristic);
  structural_health_data_Service.addCharacteristic(
    strain_gauge_data_characteristic);
  structural_health_data_Service.addCharacteristic(
    grav_gyro_data_characteristic);
  BLE.addService(structural_health_data_Service);

  BLE.advertise();
  //Serial.println("Bluetooth device active, waiting for connections...");
     // Un-comment if using serial.

  blinkTimer = millis();
  digitalWrite(redLEDPin, HIGH);
}

long startTime = -1;
/* The Loop method runs after the setup is complete.
   Method listens for BLE peripherals, and updates the values only if one
    is connected.
   Values are updated every 10 ms.
```

```
*/
void loop() {
  BLEDevice central = BLE.central();

  if (central) {
    //Serial.print("Connected to central: ");
    //Serial.println(central.address());

    digitalWrite(blueLEDPin, HIGH);  // Turn on blue LED

    while (central.connected()) {

        unsigned long currentMillis = millis();

        // Check for any characteristic subscribers
        if(strain_gauge_data_characteristic.subscribed() or
    acc_gyro_data_characteristic.subscribed() or
    grav_gyro_data_characteristic.subscribed()){
            if(startTime == -1){
              startTime = millis();
            }

             // Update sensor values when difference is larger than the
    sample hold time
            if (currentMillis - previousMillis >= sampleAndHoldTime) {
                previousMillis = currentMillis;
                ledBlink(greenLEDPin, 1); // Blink green LED

                if(strain_gauge_data_characteristic.subscribed()){
                  updateStrainGaugeValues(currentMillis - startTime);
                }
                if(acc_gyro_data_characteristic.subscribed()){
                  updateAccGyroValuesBNO(currentMillis - startTime);
                  //updateAccGyroValuesLSM(currentMillis - startTime);
                }

                if(grav_gyro_data_characteristic.subscribed()){
                  updateGravGyroValues(currentMillis - startTime);
                }
            }
        }
    }

    startTime = -1; // Reset the timer
    digitalWrite(greenLEDPin, LOW); // Turn off green LED
    digitalWrite(blueLEDPin, LOW); // Turn off blue LED
  }
}

/* Turns LED on and off every 1000 ms.
   ledPin: LED output pin, timer in milliseconds.
   timer : timer in milliseconds.
  scalar: scalar for adjusting frequency multiplicatively.
*/
void ledBlink(int ledPin, float scalar) {
  if (millis() - blinkTimer > 2000 * scalar) {
    digitalWrite(ledPin, HIGH);
    blinkTimer = millis();
  }
  else if (millis() - blinkTimer > 1000 * scalar) {
    digitalWrite(ledPin, LOW);
  }
}

// Reads the strain gauge sensors, stores them, and writes them to the Bluetooth characteristic.
// The bluetooth characteristic publishes on write.
void updateStrainGaugeValues(float timeStamp) {
  strain_gauge_data.timer = timeStamp;
  strain_gauge_data.gauge1 = analogRead(A0);
  strain_gauge_data.gauge2 = analogRead(A1);
  strain_gauge_data.gauge3 = analogRead(A2);
  strain_gauge_data.gauge4 = analogRead(A3);
  strain_gauge_data.gauge5 = analogRead(A4);
  strain_gauge_data.gauge8 = analogRead(A5);
  strain_gauge_data_characteristic.writeValue(strain_gauge_data.bytes, 16); //sizeof(strain_gauge_data)
}


// Reads the acceleration and gyroscope of the BNO055 IMU, stores them, and writes them to the Bluetooth characteristic.
// The bluetooth characteristic publishes on write.
void updateGravGyroValues(float timeStamp) {
    IMUSensor.updateGyro();
    IMUSensor.updateGravAccel();          //Update the Gravity Acceleration data
    IMUSensor.updateCalibStatus();
    grav_gyro_data.timer = timeStamp;
    grav_gyro_data.gyro_y = IMUSensor.readGyroX() * 100;
    grav_gyro_data.acc_x = IMUSensor.readGravAcceleration(X_AXIS) * 100;
    grav_gyro_data.acc_z  = IMUSensor.readGravAcceleration(Z_AXIS) * 100;
    grav_gyro_data_characteristic.writeValue(grav_gyro_data.bytes, 10); //sizeof(grav_gyro_data)
}


float ax,ay,az;
float gyx,gyy,gyz;
// Reads the acceleration and gyroscope of the LSM IMU, stores them, and writes them to the Bluetooth characteristic.
// The bluetooth characteristic publishes on write.
void updateAccGyroValuesLSM(float timeStamp) {
    IMU.readGyroscope(gyx, gyy, gyz);
    IMU.readAcceleration(ax, ay, az);
    acc_gyro_data.timer = timeStamp;
    acc_gyro_data.acc_x = short(ax*100);
    acc_gyro_data.acc_z = short(az*100);
    acc_gyro_data.gyro_y = short(gyy*100);
    acc_gyro_data_characteristic.writeValue(acc_gyro_data.bytes, 10); // sizeof(grav_gyro_data)
}

// Updates the IMUs, reads the acceleration and gyroscope of the BNO055 IMU, stores them, and writes them to the Bluetooth characteristic.
// The bluetooth characteristic publishes on write.
void updateAccGyroValuesBNO(float timeStamp) {
    IMUSensor.updateGyro();
    IMUSensor.updateAccel();          //Update the Gravity Acceleration data
    IMUSensor.updateCalibStatus();
    acc_gyro_data.timer = timeStamp;
    acc_gyro_data.gyro_y = IMUSensor.readGyroX() * 100;
    acc_gyro_data.acc_x = IMUSensor.readAccelerometer(X_AXIS) * 100;
    acc_gyro_data.acc_z  = IMUSensor.readAccelerometer(Z_AXIS) * 100;
    acc_gyro_data_characteristic.writeValue(acc_gyro_data.bytes, 10); // sizeof(grav_gyro_data)
}
}
```

Appendix D

Mendix Pages



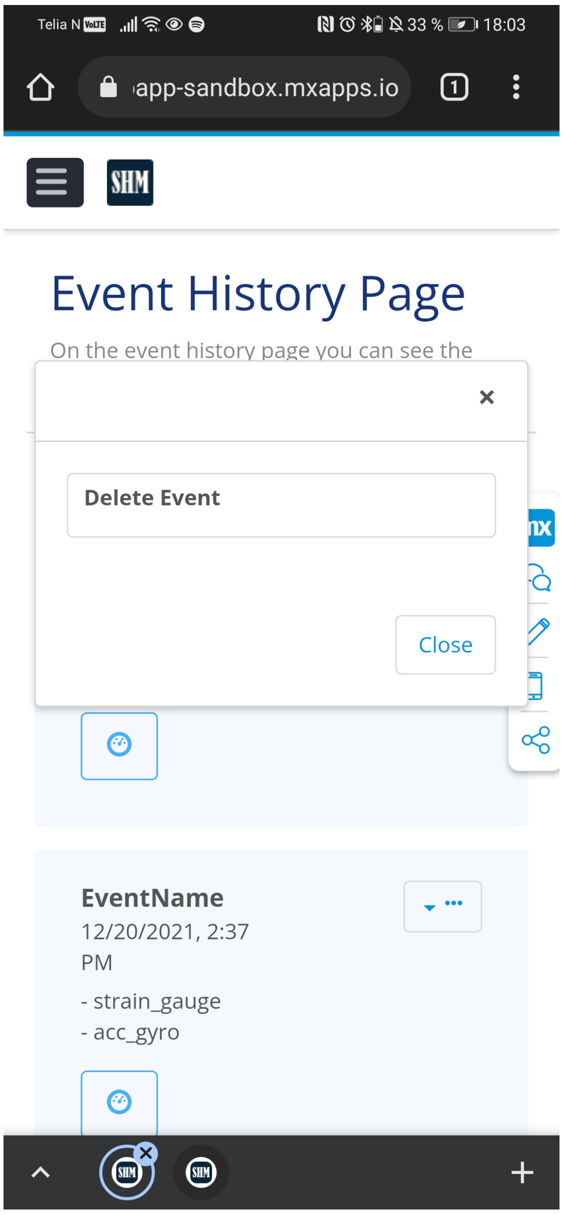
Event History Page

On the event history page you can see the events logged by you.

Liten hump Strindheim
12/20/2021, 2:44 PM
- strain_gauge
- acc_gyro

EventName
12/20/2021, 2:37 PM
- strain_gauge
- acc_gyro

Fig. D.1. Event history page for seeing events that are logged.

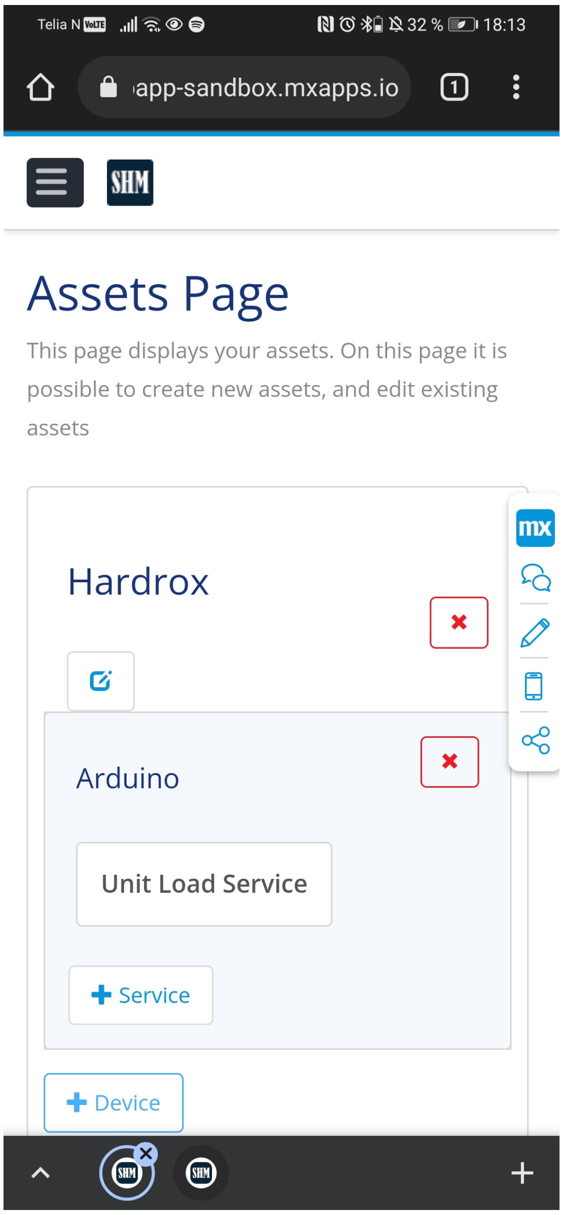


Event History Page

On the event history page you can see the

EventName
12/20/2021, 2:37 PM
- strain_gauge
- acc_gyro

Fig. D.2. Event history page overlay for deleting event.


Telia N 32 % 18:13

app-sandbox.mxapps.io

SHM

Assets Page

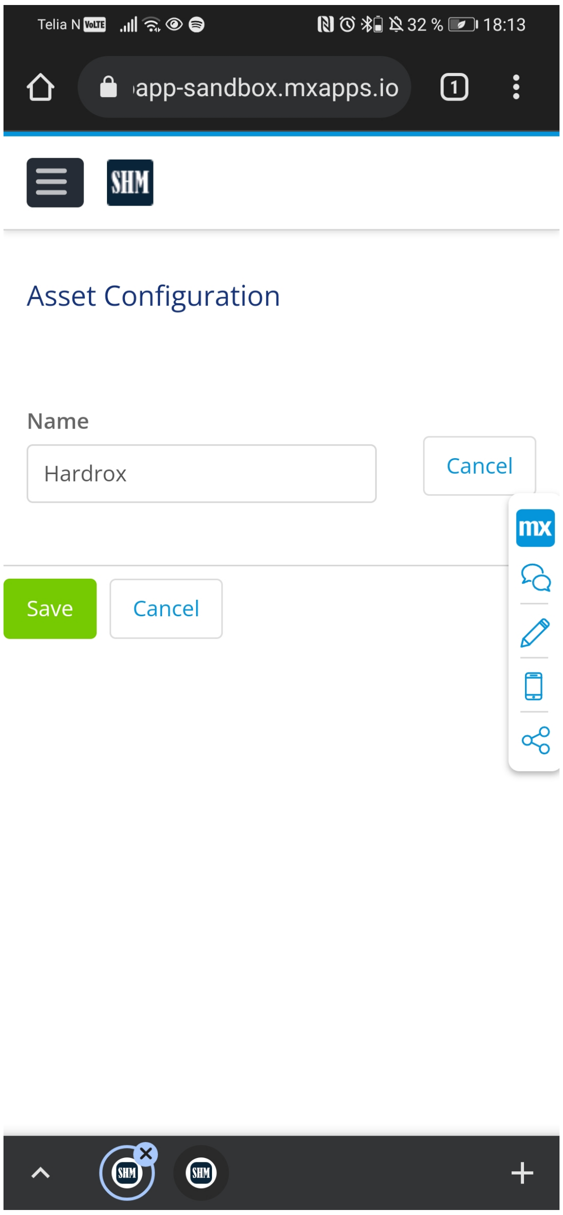
This page displays your assets. On this page it is possible to create new assets, and edit existing assets

Hardrox

Arduino

Unit Load Service

Fig. D.3. Asset page.


Telia N 32 % 18:13

app-sandbox.mxapps.io

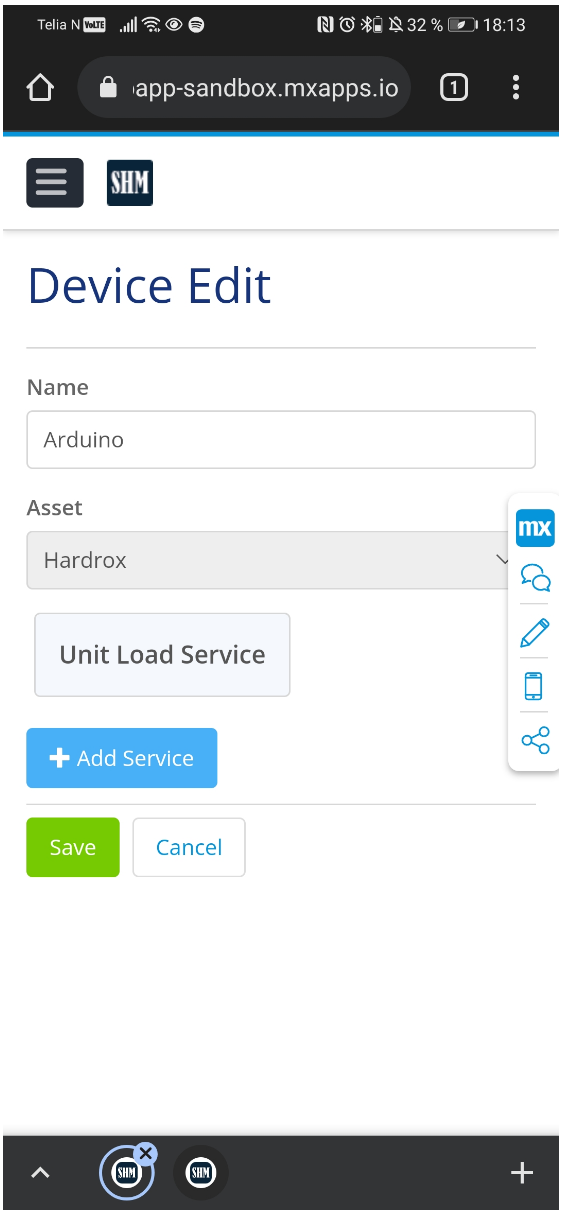
SHM

Asset Configuration

Name

Hardrox

Fig. D.4. Asset configuration.

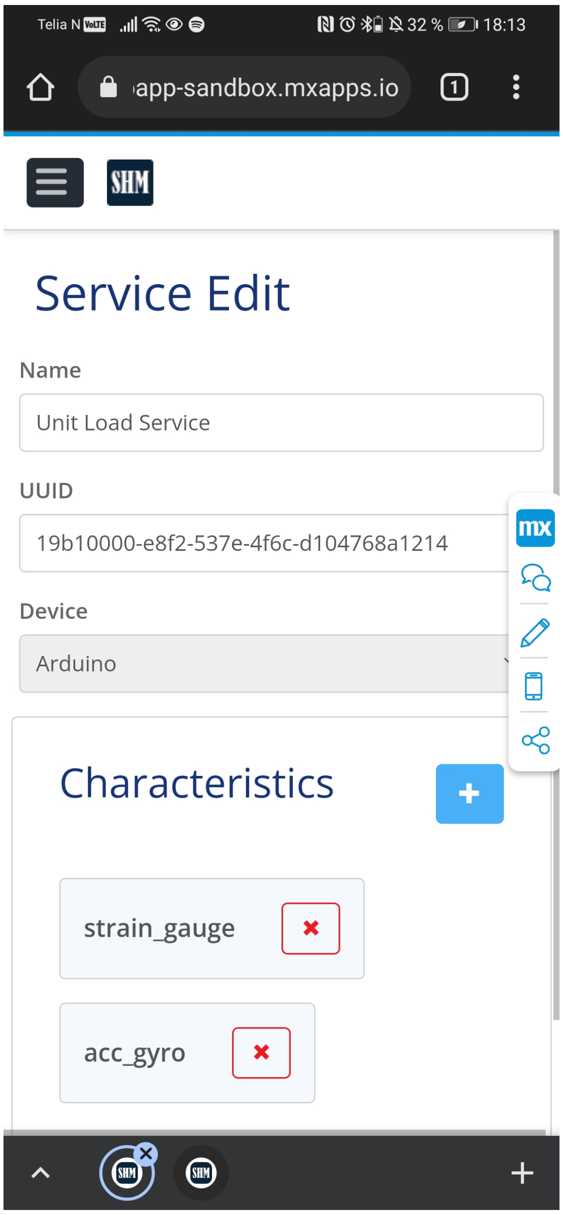

Device Edit

Name: Arduino

Asset: Hardrox

Unit Load Service

Fig. D.5. Device configuration.


Service Edit

Name: Unit Load Service

UUID: 19b10000-e8f2-537e-4f6c-d104768a1214

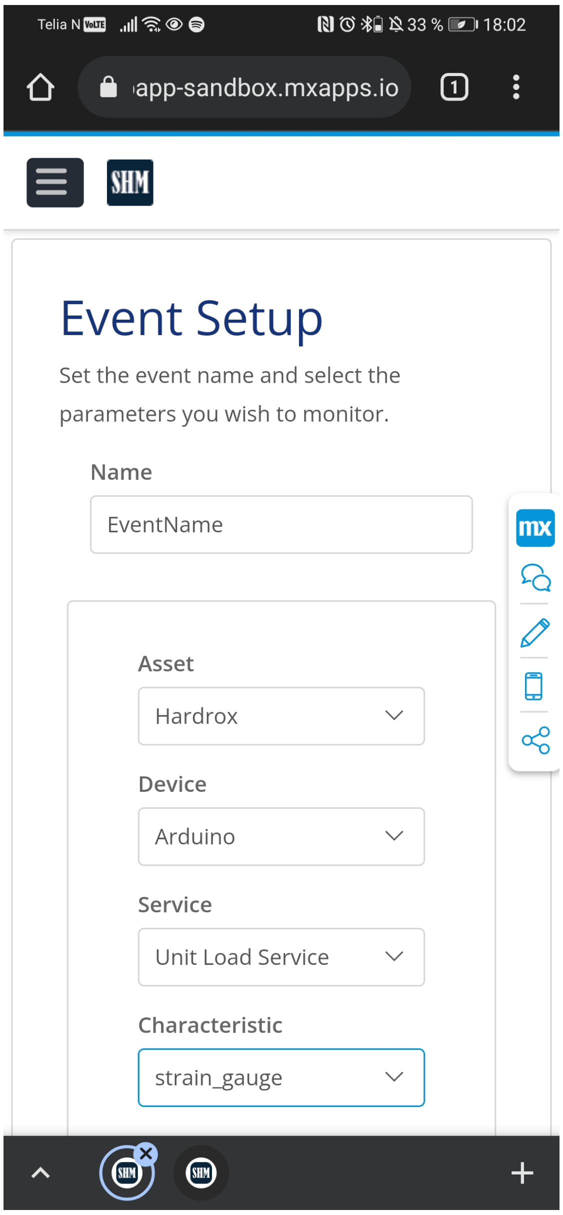
Device: Arduino

Characteristics

strain_gauge

acc_gyro

Fig. D.6. Service configuration.



Telia N 33 % 18:02

app-sandbox.mxapps.io

SHM

Event Setup

Set the event name and select the parameters you wish to monitor.

Name: EventName

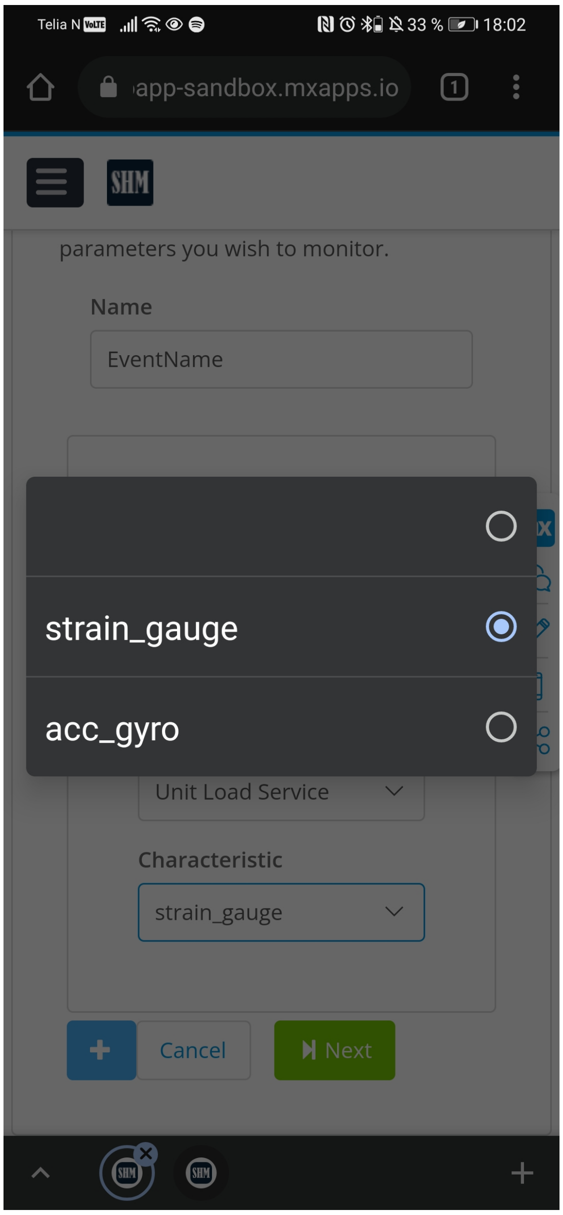
Asset: Hardrox

Device: Arduino

Service: Unit Load Service

Characteristic: strain_gauge

Fig. D.7. Event Setup.



Telia N 33 % 18:02

app-sandbox.mxapps.io

SHM

parameters you wish to monitor.

Name: EventName

( )

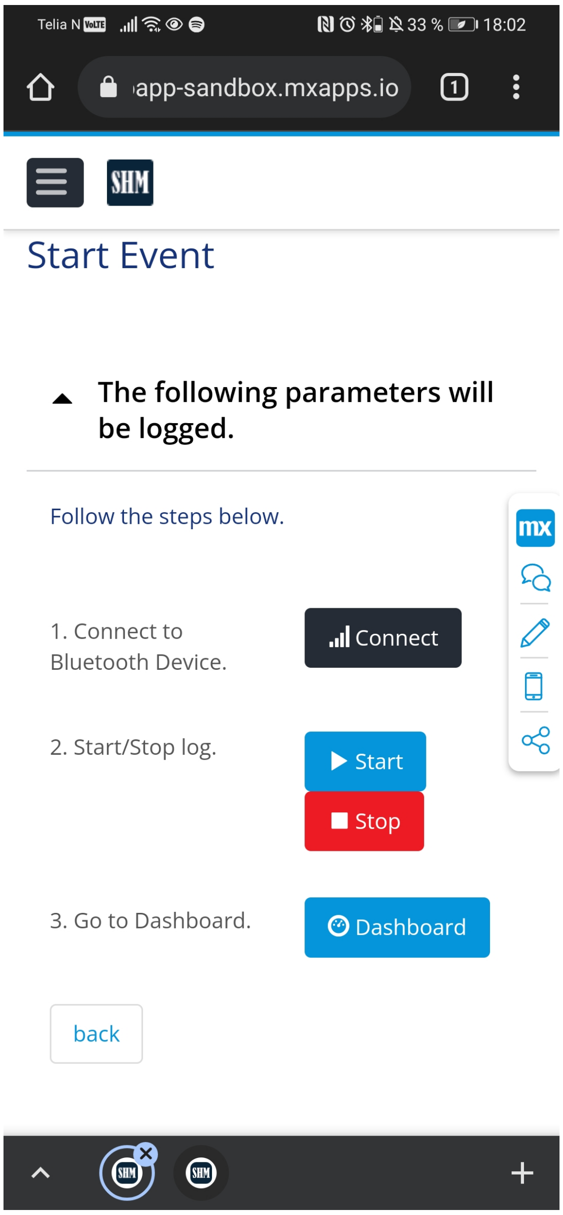
(●) strain_gauge

( ) acc_gyro

Unit Load Service

Characteristic: strain_gauge

Fig. D.8. Event Setup Overlay.


Telia N 33 % 18:02

app-sandbox.mxapps.io

Start Event

The following parameters will be logged.

Follow the steps below.

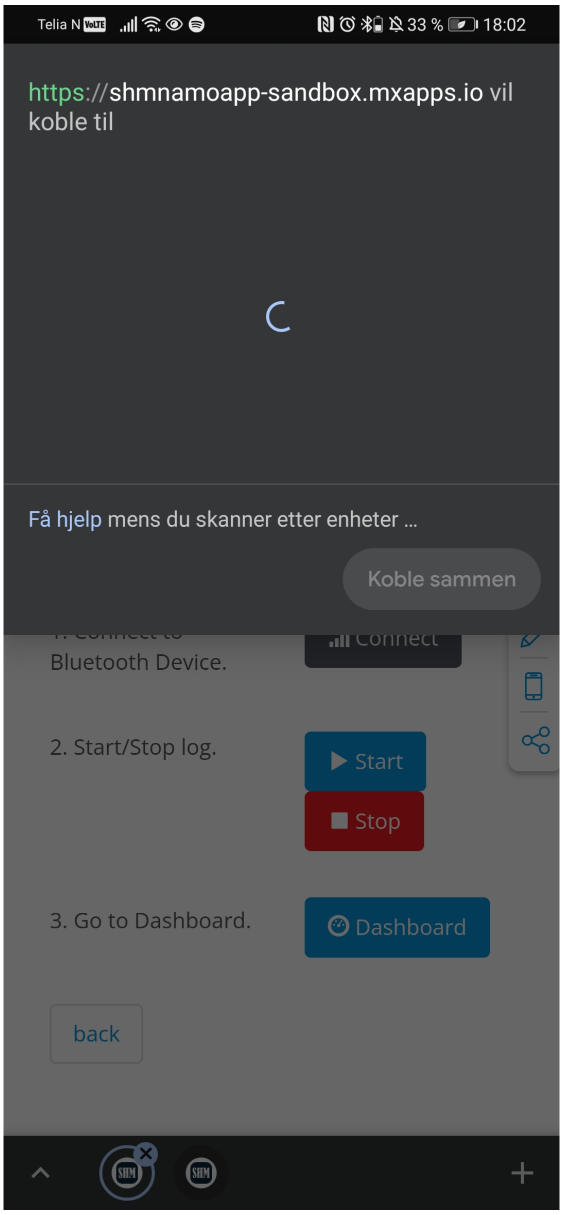
1. Connect to Bluetooth Device. Connect

2. Start/Stop log. Start Stop

3. Go to Dashboard. Dashboard

back

Fig. D.9. Event Start Page.

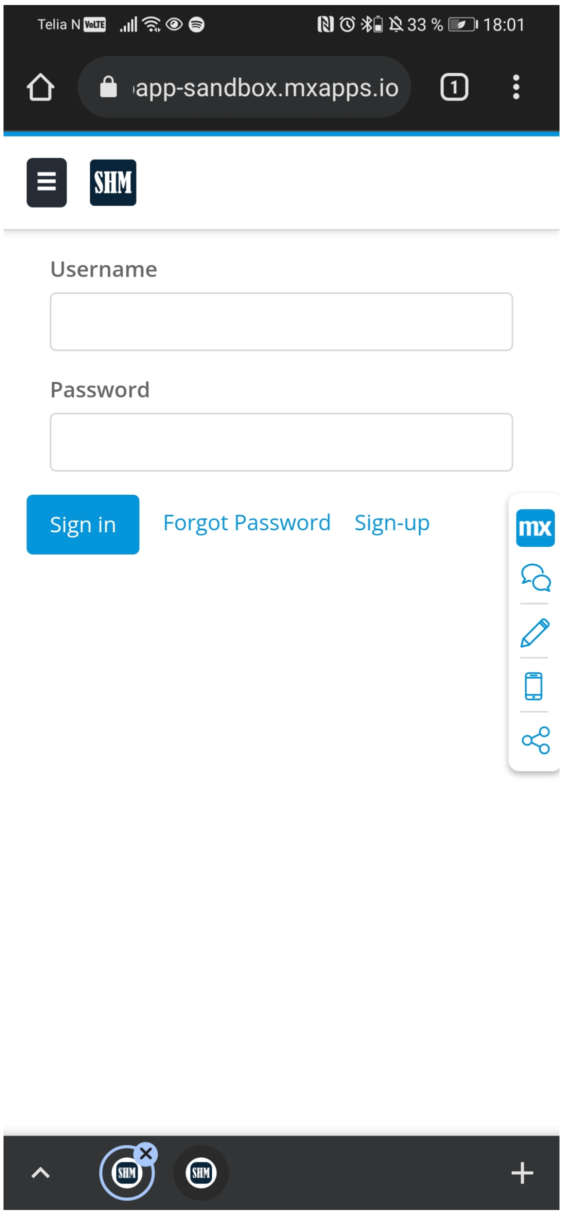

Telia N 33 % 18:02

https://shmnamoapp-sandbox.mxapps.io vil koble til

Få hjelp mens du skanner etter enheter ...

Koble sammen

Fig. D.10. Event Start Bluetooth.


Telia N 33 % 18:01

app-sandbox.mxapps.io

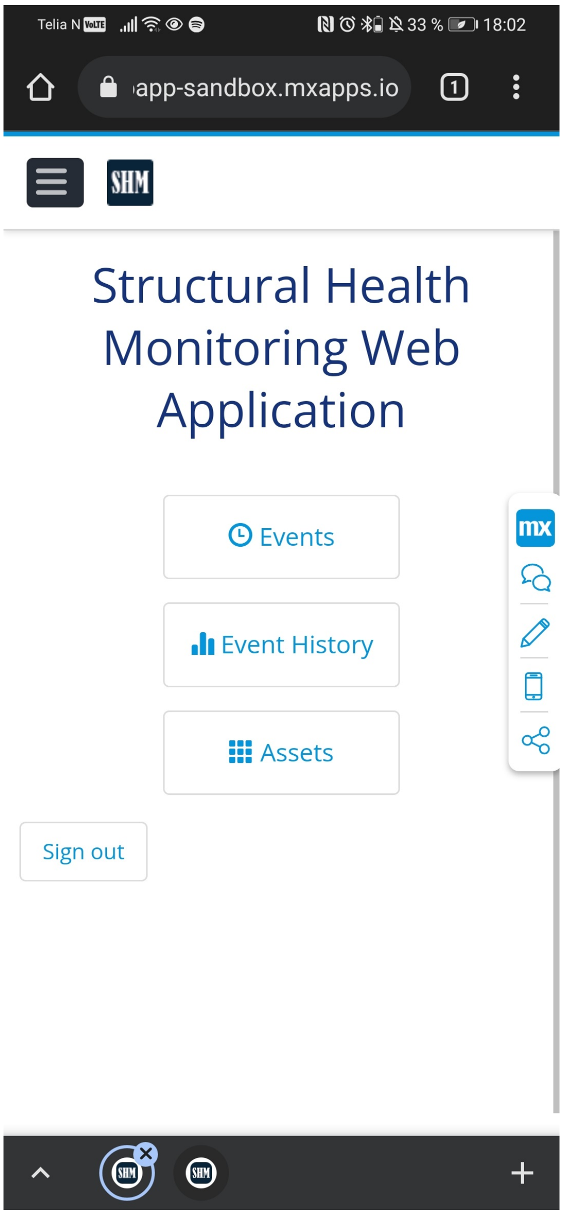
SHM

Username

Password

Sign in Forgot Password Sign-up

Fig. D.11. Event Setup.


Telia N 33 % 18:02

app-sandbox.mxapps.io

SHM

Structural Health Monitoring Web Application

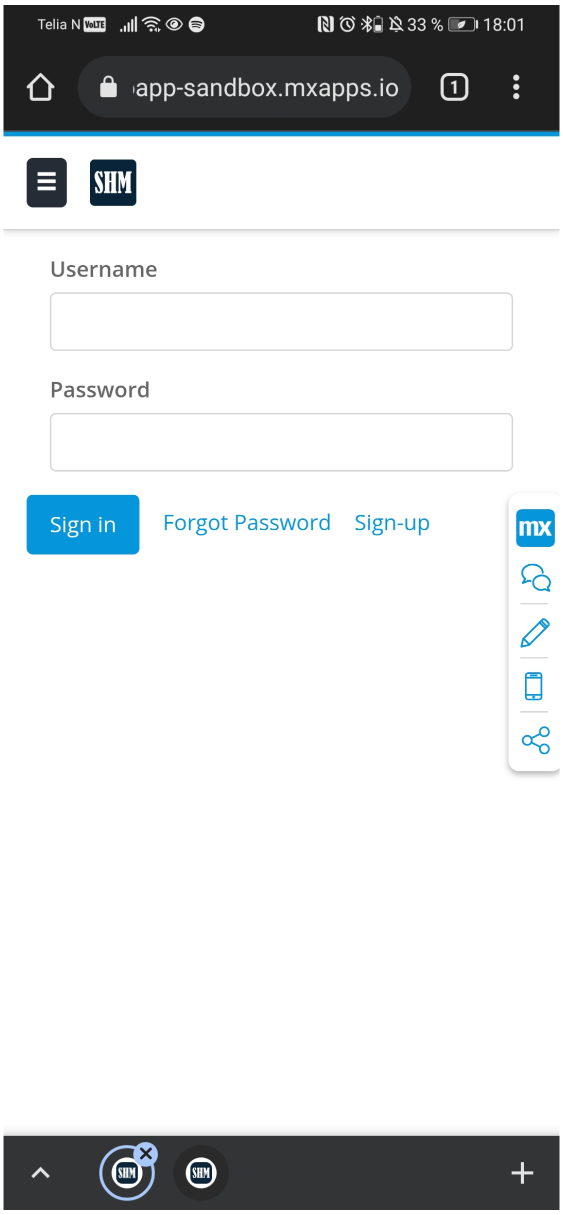
Events

Event History

Assets

Sign out

Fig. D.12. Event Setup Overlay.


Telia N 33 % 18:01

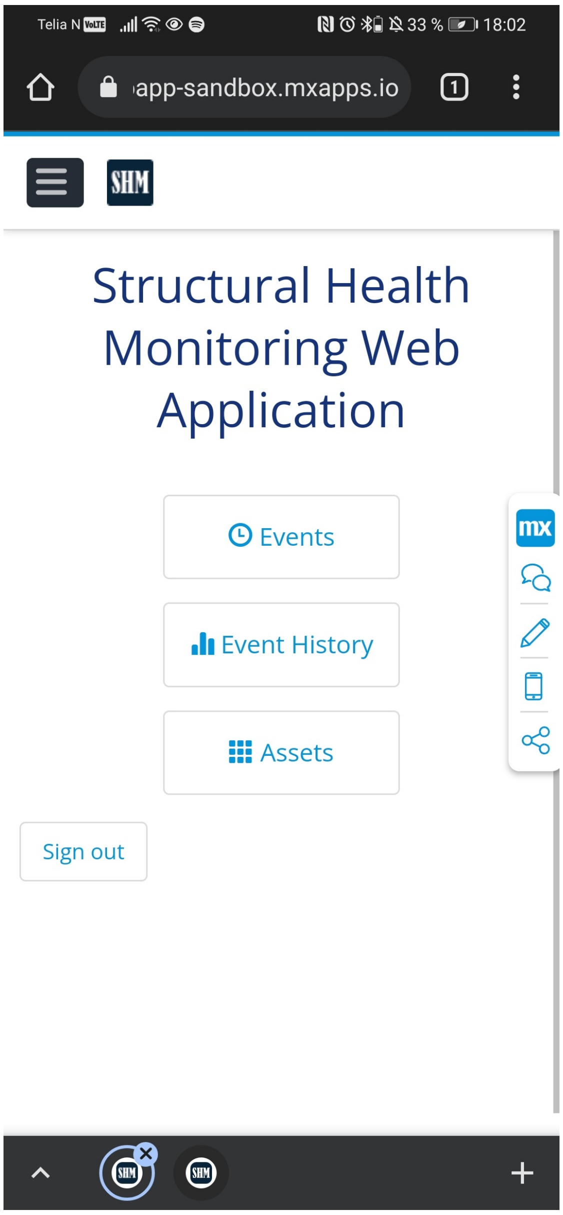
app-sandbox.mxapps.io

Username

Password

Sign in Forgot Password Sign-up

Fig. D.13. Event Setup.


Telia N 33 % 18:02

app-sandbox.mxapps.io

Structural Health Monitoring Web Application

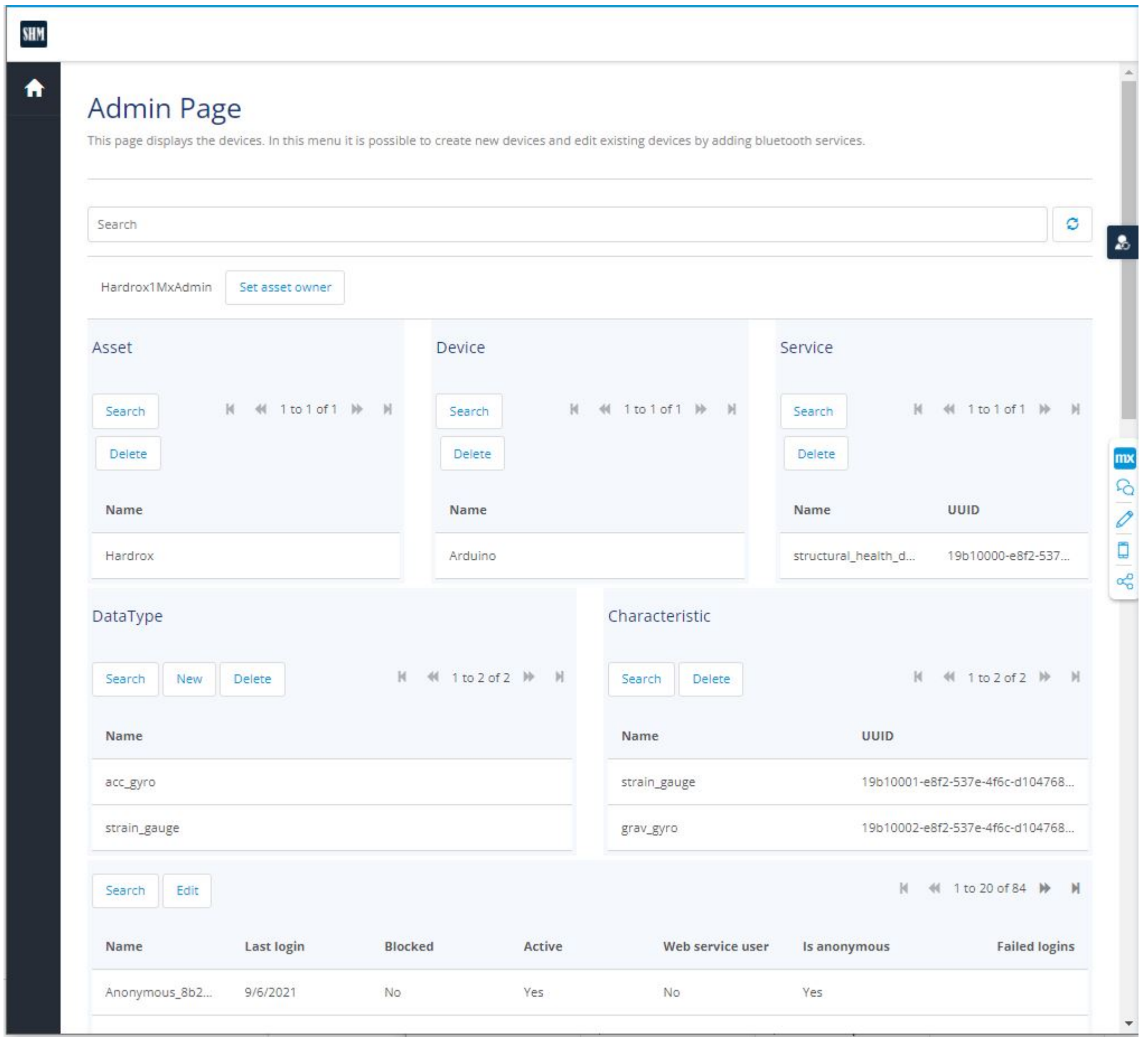
Events

Event History

Assets

Sign out

Fig. D.14. Event Setup Overlay.



Admin Page

This page displays the devices. In this menu it is possible to create new devices and edit existing devices by adding bluetooth services.

Hardrox1MxAdmin

Asset — 1 to 1 of 1

Name
Hardrox

Device — 1 to 1 of 1

Name
Arduino

Service — 1 to 1 of 1

Name	UUID
structural_health_d...	19b10000-e8f2-537...

DataType — 1 to 2 of 2

Name
acc_gyro
strain_gauge

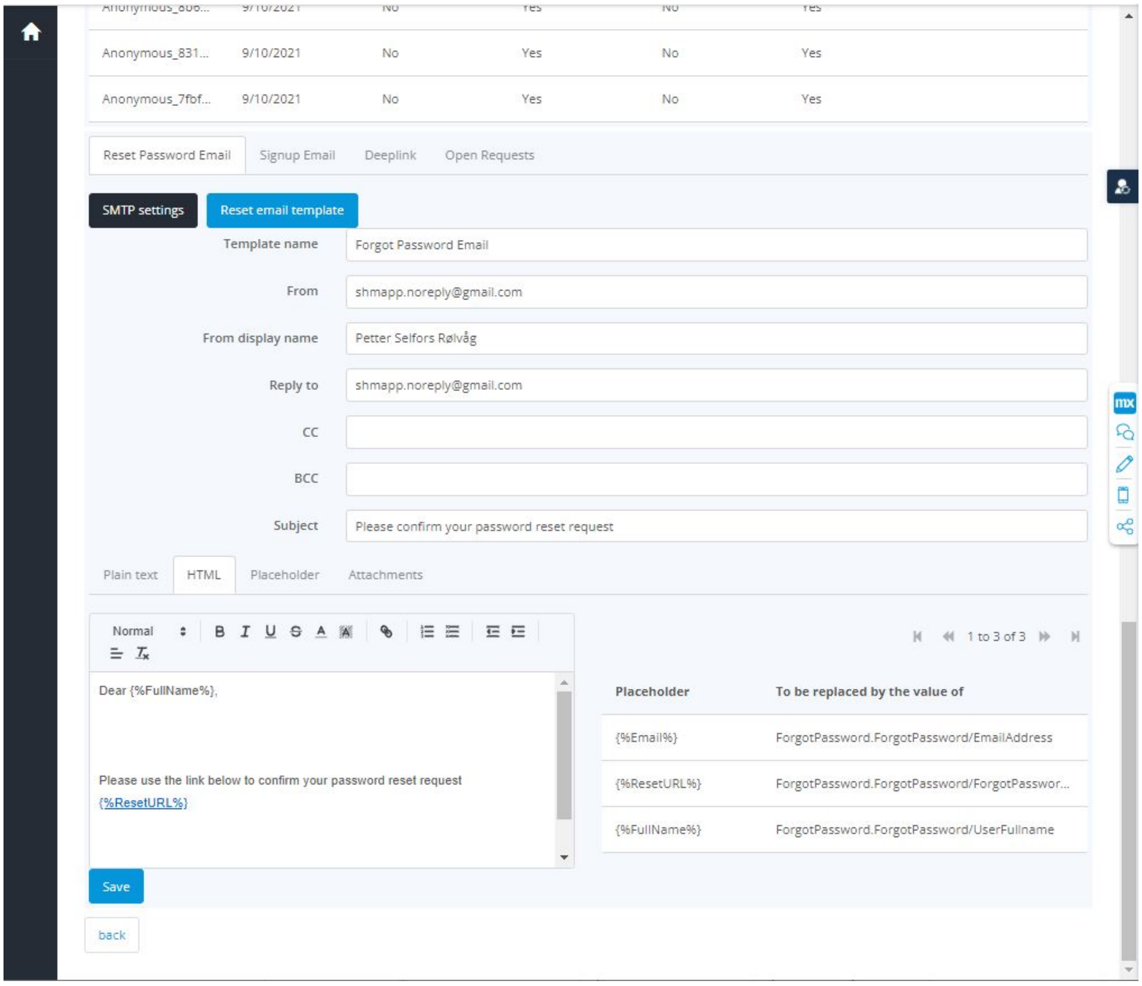
Characteristic — 1 to 2 of 2

Name	UUID
strain_gauge	19b10001-e8f2-537e-4f6c-d104768...
grav_gyro	19b10002-e8f2-537e-4f6c-d104768...

1 to 20 of 84

Name	Last login	Blocked	Active	Web service user	Is anonymous	Failed logins
Anonymous_8b2...	9/6/2021	No	Yes	No	Yes	

Fig. D.15. Admin Page



Anonymous_831...	9/10/2021	No	Yes	No	Yes
Anonymous_7fbf...	9/10/2021	No	Yes	No	Yes

Template name	Forgot Password Email
From	shmapp.noreply@gmail.com
From display name	Petter Selfors Rølvåg
Reply to	shmapp.noreply@gmail.com
CC	
BCC	
Subject	Please confirm your password reset request

Dear {%FullName%},

Please use the link below to confirm your password reset request
{%ResetURL%}

Placeholder	To be replaced by the value of
{%Email%}	ForgotPassword.ForgotPassword/EmailAddress
{%ResetURL%}	ForgotPassword.ForgotPassword/ForgotPasswor...
{%FullName%}	ForgotPassword.ForgotPassword/UserFullname

Fig. D.16. Admin Page 2

Appendix E

Mendix Documentation

App

Module 'SHM_Module'

Domain model

Entities

Name	Generalization	Documentation
acc_gyro	SHM_Module.Data	Holds acc_gyro data coming from the Arduino Uno Wifi Rev 2 microcontroller.
Asset		An asset is considered as the physical asset which is to be monitored in the system.
Characteristic		The entity represents a Bluetooth characteristic. It holds information about the name and UUID of the characteristic.
ClientData		The ClientData entity holds a data string of unlimited length for sending sensor data from the devices to the server.
Data		The data Entity is the super entity for other data types that the database supports. It is possible for other data types to generalize (inherit) from this object and be connected with an Event. This object should have no data of any kind and should always be inherited.
DataType		The DataType entity is used to store the names of the data kinds that are present in the system. The key-sensitive name attribute refers to a specific entity in the domain model for triggering certain functionalities at runtime. This is a required component for the system and must be utilized correctly for the system to be deployed effectively. DataTypes can only be created by the administrator through the admin page. For further insight, see the Javascript or Java code.
Device		The Device entity represents a Bluetooth device, such as an Arduino microcontroller. It holds information about the name of the device.

Event		One Event entity is created for each logged characteristic.
EventBatch		One EventBatch entity is created for keeping track of the logged Events. The EventBatch can be associated with multiple Events.
EventSettings		The EventSettings is used to enter custom settings for the user. Currently, it holds the value of the biker's mass in order to use it as a part of the inverse method.
float32	SHM_Module.Data	Holds 4 byte float data coming from the Arduino Uno Wifi Rev 2 microcontroller.
grav_gyro	SHM_Module.Data	Holds acc_gyro data coming from the Arduino Uno Wifi Rev 2 microcontroller.
inverse_data		Holds the inverse data which is associated to a specific acc_gyro.
Login		The Login Entity is used for storing user credentials when the user attemps to login.
Service		The Service entity represents a Bluetooth service. It holds information about the bluetooth service name and UUID.
strain_gauge	SHM_Module.Data	Holds strain gauge data coming from the Arduino Uno Wifi Rev 2 microcontroller.
uint	SHM_Module.Data	Holds 2 byte uint data coming from the Arduino Uno Wifi Rev 2 microcontroller.

Entity 'acc_gyro'

Holds acc_gyro data coming from the Arduino Uno Wifi Rev 2 microcontroller.

Generalization

SHM_Module.Data

Attributes

Name	Type	Default value	Documentation
Timer	Decimal	0	The timestamp of when the sensor is read.

acc_x	Decimal	0	Acceleration read from IMU in x direction.
acc_z	Decimal	0	Acceleration read from IMU in z direction.
gyro_y	Decimal	0	Pitch rate read from IMU in y direction.

Associations

Name	Connected to	Multiplicity	Documentation
inverse_data_acc_gyro	SHM_Module.inverse_data	One-to-one	The association keeps track of the inverse data that corresponds with a given acc_gyro entity. One inverse_data should refer to a single acc_gyro entity.

Entity 'Asset'

An asset is considered as the physical asset which is to be monitored in the system.

Attributes

Name	Type	Default value	Documentation
Name	String (100)		The name of the asset.

Associations

Name	Connected to	Multiplicity	Documentation
Asset_User	System.User	One-to-many	This association is put in so that the user can only retrieve assets that themselves created. A user can be associated with multiple assets.

Entity 'Characteristic'

The entity represents a Bluetooth characteristic. It holds information about the name and UUID of the characteristic.

Attributes

Name	Type	Default value	Documentation
Name	String (32)		The name of the bluetooth characteristic.
UUID	String (36)		The unique identifier for the characteristic.

Associations

Name	Connected to	Multiplicity	Documentation
Characteristic_DataType	SHM_Module.DataType	One-to-many	The association keeps track of a characteristic's data type.
Characteristic_Service	SHM_Module.Service	One-to-many	A characteristic can only be linked to one Service. A Service can have many Characteristics.

Entity 'ClientData'

The ClientData entity holds a data string of unlimited length for sending sensor data from the devices to the server.

Attributes

Name	Type	Default value	Documentation
data	String (unlimited)		Keeps hold of the data string which is being sent to the server. The data string holds the sensor values read by the microcontroller.

Associations

Entity 'ClientData' does not own any associations.

Entity 'Data'

The data Entity is the super entity for other data types that the database supports. It is possible for other data types to generalize (inherit) from this object and be connected with an Event. This object should have no data of any kind and should always be inherited.

Attributes

Entity 'Data' has no attributes.

Associations

Name	Connected to	Multiplicity	Documentation
Data_Event	SHM_Module.Event	One-to-many	The association is used to retrieve different data types from an Event. The association is inherited by the Data child entities, which results in a cleaner database.

Entity 'DataType'

The DataType entity is used to store the names of the data kinds that are present in the system. The key-sensitive name attribute refers to a specific entity in the domain model for triggering certain functionalities at runtime. This is a required component for the system and must be utilized correctly for the system to be deployed effectively. DataTypes can only be created by the administrator through the admin page. For further insight, see the Javascript or Java code.

Attributes

Name	Type	Default value	Documentation
Name	String (50)		

Associations

Entity 'DataType' does not own any associations.

Entity 'Device'

The Device entity represents a Bluetooth device, such as an Arduino microcontroller. It holds information about the name of the device.

Attributes

Name	Type	Default value	Documentation

Name	String (100)		The name of the device.

Associations

Name	Connected to	Multiplicity	Documentation
Device_Asset	SHM_Module.Asset	One-to-many	A device can only be linked to one asset. An Asset can have many devices.

Entity 'Event'

One Event entity is created for each logged characteristic.

Attributes

Name	Type	Default value	Documentation
Name	String (20)		The name of the event.

Associations

Name	Connected to	Multiplicity	Documentation
Event_Asset	SHM_Module.Asset	One-to-many	An arbitrary amount of Assets can be logged in an event.
Event_Characteristic	SHM_Module.Characteristic	One-to-many	
Event_Device	SHM_Module.Device	One-to-many	An arbitrary amount of devices can be logged in an event.
Event_EventBatch	SHM_Module.EventBatch	One-to-many	
Event_Service	SHM_Module.Service	One-to-many	An arbitrary amount of Services can be logged in an

			event.

Entity 'EventBatch'

One EventBatch entity is created for keeping track of the logged Events. The EventBatch can be associated with multiple Events.

Attributes

Name	Type	Default value	Documentation
Name	String (200)	EventName	The name of the event
IsLogging	Boolean	false	Value to keep track of when the event is logged by the rider. When this is true, there is an ongoing event where the rider is logging sensor values from the bike.
IsDeviceConnected	Boolean	false	

Associations

Name	Connected to	Multiplicity	Documentation
EventSettings_EventBatch	SHM_Module.EventSettings	One-to-one	Settings for the eventbatch.

Entity 'EventSettings'

The EventSettings is used to enter custom settings for the user. Currently, it holds the value of the biker's mass in order to use it as a part of the inverse method.

Attributes

Name	Type	Default value	Documentation
Mass	Decimal	0	The mass of the bicycle rider

Associations

Name	Connected to	Multiplicity	Documentation
EventSettings_EventBatch	SHM_Module.EventBatch	One-to-one	Settings for the eventbatch.

Entity 'float32'

Holds 4 byte float data coming from the Arduino Uno Wifi Rev 2 microcontroller.

Generalization

SHM_Module.Data

Attributes

Name	Type	Default value	Documentation
Value	Decimal	0	Holds the value of the float 32.

Associations

Entity 'float32' does not own any associations.

Entity 'grav_gyro'

Holds acc_gyro data coming from the Arduino Uno Wifi Rev 2 microcontroller.

Generalization

SHM_Module.Data

Attributes

Name	Type	Default value	Documentation
Timer	Decimal	0	The timestamp of when the sensor is read.
grav_x	Decimal	0	Gravitational acceleration in x direction read from IMU.
grav_z	Decimal	0	Gravitational acceleration in z direction read from IMU.
gyro_y	Decimal	0	Pitch rate read from IMU in y direction.

Associations

Entity 'grav_gyro' does not own any associations.

Entity 'inverse_data'

Holds the inverse data which is associated to a specific acc_gyro.

Attributes

Name	Type	Default value	Documentation
gauge1	Decimal	0	
gauge2	Decimal	0	
gauge3	Decimal	0	
gauge4	Decimal	0	
gauge5	Decimal	0	
gauge8	Decimal	0	

Associations

Name	Connected to	Multiplicity	Documentation
inverse_data_acc_gyro	SHM_Module.acc_gyro	One-to-one	The association keeps track of the inverse data that corresponds with a given acc_gyro entity. One inverse_data should refer to a single acc_gyro entity.

Entity 'Login'

The Login Entity is used for storing user credentials when the user attemps to login.

Attributes

Name	Type	Default value	Documentation
Username	String (200)		Holds the username of the user who is attempting to login.
Password	String (200)		Holds the password of the user who is attempting to login.
ValidationMessage	String (unlimited)		Validation message to inform the user of login errors.

Associations

Entity 'Login' does not own any associations.

Entity 'Service'

The Service entity represents a Bluetooth service. It holds information about the bluetooth service name and UUID.

Attributes

Name	Type	Default value	Documentation
Name	String (100)		The name of the service.
UUID	String (200)		The unique identifier for the bluetooth service.

Associations

Name	Connected to	Multiplicity	Documentation
Service_Device	SHM_Module.Device	One-to-many	A service can only be linked to one device. A device can have many services.

Entity 'strain_gauge'

Holds strain gauge data coming from the Arduino Uno Wifi Rev 2 microcontroller.

Generalization

SHM_Module.Data

Attributes

Name	Type	Default value	Documentation
Timer	Decimal	0	The timestamp of when the sensor is read.
gauge1	Integer	0	Value of strain gauge 1 mounted on the Hardrocx bicycle.
gauge2	Integer	0	Value of strain gauge 2 mounted on the Hardrocx bicycle.
gauge3	Integer	0	Value of strain gauge 3 mounted on the Hardrocx bicycle.

gauge4	Integer	0	Value of strain gauge 4 mounted on the Hardrocx bicycle.
gauge5	Integer	0	Value of strain gauge 5 mounted on the Hardrocx bicycle.
gauge8	Integer	0	Value of strain gauge 8 mounted on the Hardrocx bicycle.

Associations

Entity 'strain_gauge' does not own any associations.

Entity 'uint'

Holds 2 byte uint data coming from the Arduino Uno Wifi Rev 2 microcontroller.

Generalization

SHM_Module.Data

Attributes

Name	Type	Default value	Documentation
Value	Integer	0	Holds the value of the uint.

Associations

Entity 'uint' does not own any associations.

Enumerations

The module has no enumerations.

Microflows

Name	Return type	Documentation
AddDevice		Creates a new device and associates it to the input asset. The device is commited and refreshed for the client.
AddEvent		Creates a new Event and associates it to the input EventBatch. The new Event is commited and refreshed for the client.
AddService		Creates a new Service and associates it

		to the input Device. The new Service is commited and refreshed for the client.
CalculateInverseData		Each time an acc gyro object is committed to the database, the microflow is executed. It performs the inverse method on the gathered data every four seconds.
CreateAsset		Creates an Asset, commits the Asset and updates the client.
CreateCharacteristic		Creates and returns a Characteristic Entity.
CreateEventBatch		Creates an EventBatch, an Event and returns the EventBatch.
ExitSetupEvent		Exits the event setup page by opening the home page and deleting the Event Batch.
GetAccGyroData		Returns a list of acc_gyro data.
GetAccGyroDataSorted		Returns a list of sorted acc_gyro data.
GetEvent		Gets the head of the events associated to an EventBatch.
GetEventSetting		Gets or creates the EventSettings from an EventBatch.
GetEventsReversed		Returns a list of events sorted by createdDate in descending order.
GetGravGyroDataSorted		Returns a list of sorted acc_gyro data.
GetInverseDataSorted		Returns a list of sorted inverse data in ascending order of the timer..
GetStrainGaugeData		Returns a list of strain gauge data.
GetStrainGaugeDataSorted		Returns a list of sorted strain gauge data.
GetUserAssets		Retrieves a list of the Assets associated to the current user.
OpenEventStartPage		Opens the event start page.
OpenHomepage		This is the first microflow which is run as

	the client enters the web site. It executes a deep link and opens the sign in page whether the deeplink is executed or not.
RegisterUser_m	Registers a user.
SelectableCharacteristics	Retrieves a list of selectable characteristics given an event.
SelectableDevices	Retrieves a list of selectable devices given an event.
SelectableServices	Retrieves a list of selectable services given an event.
SendDataToServer	Sends a datastring to the server, parses, commits, and deletes the data.
SetAssetOwner	Sets the current user as the owner of the given Asset.

Microflow 'AddDevice'

Creates a new device and associates it to the input asset. The device is commited and refreshed for the client.

Parameters

Name	Type	Documentation
Asset	SHM_Module.Asset	The asset which will be associated with the new device.

Return type

Nothing

Microflow 'AddEvent'

Creates a new Event and associates it to the input EventBatch. The new Event is commited and refreshed for the client.

Parameters

Name	Type	Documentation
EventBatch	SHM_Module.EventBatch	The EventBatch which will be associated with the new Event.

Return type

Nothing

Microflow 'AddService'

Creates a new Service and associates it to the input Device. The new Service is commited and refreshed for the client.

Parameters

Name	Type	Documentation
Device	SHM_Module.Device	The Device which will be associated with the new Service.

Return type

Nothing

Microflow 'CalculateInverseData'

Each time an acc gyro object is committed to the database, the microflow is executed. It performs the inverse method on the gathered data every four seconds.

Parameters

Name	Type	Documentation
acc_gyro	SHM_Module.acc_gyro	The acc_gyro entity which holds the timer value to check whether 4 seconds of data is accumulated.

Return type

Nothing

Microflow 'CreateAsset'

Creates an Asset, commits the Asset and updates the client.

Parameters

This microflow has no parameters.

Return type

Nothing

Microflow 'CreateCharacteristic'

Creates and returns a Characteristic Entity.

Parameters

Name	Type	Documentation
Service	SHM_Module.Service	The Service which will be associated with the new Characteristic.

Return type

SHM_Module.Characteristic

Microflow 'CreateEventBatch'

Creates an EventBatch, an Event and returns the EventBatch.

Parameters

This microflow has no parameters.

Return type

SHM_Module.EventBatch

Microflow 'ExitSetupEvent'

Exits the event setup page by opening the home page and deleting the Event Batch.

Parameters

Name	Type	Documentation
EventBatch	SHM_Module.EventBatch	EventBatch to be deleted.

Return type

Nothing

Microflow 'GetAccGyroData'

Returns a list of acc_gyro data.

Parameters

Name	Type	Documentation
Event	SHM_Module.Event	Event used to get all associated acc_gyro entities.

Return type

List of SHM_Module.acc_gyro

Microflow 'GetAccGyroDataSorted'

Returns a list of sorted acc_gyro data.

Parameters

Name	Type	Documentation
Event	SHM_Module.Event	Event used to get all associated acc_gyro entities.

Return type

List of SHM_Module.acc_gyro

Microflow 'GetEvent'

Gets the head of the events associated to an EventBatch.

Parameters

Name	Type	Documentation
EventBatch	SHM_Module.EventBatch	EventBatch used to get the assciated events.

Return type

SHM_Module.Event

Microflow 'GetEventSetting'

Gets or creates the EventSettings from an EventBatch.

Parameters

Name	Type	Documentation
EventBatch	SHM_Module.EventBatch	EventBatch used to get the associated EventSetting

Return type

SHM_Module.EventSettings

Microflow 'GetEventsReversed'

Returns a list of events sorted by createdDate in descending order.

Parameters

This microflow has no parameters.

Return type

List of SHM_Module.EventBatch

Microflow 'GetGravGyroDataSorted'

Returns a list of sorted acc_gyro data.

Parameters

Name	Type	Documentation
Event	SHM_Module.Event	Used to get all associated grav_gyro entities.

Return type

List of SHM_Module.grav_gyro

Microflow 'GetInverseDataSorted'

Returns a list of sorted inverse data in ascending order of the timer..

Parameters

Name	Type	Documentation
Event	SHM_Module.Event	Used to get all associated acc_gyro entities.

Return type

List of SHM_Module.inverse_data

Microflow 'GetStrainGaugeData'

Returns a list of strain gauge data.

Parameters

Name	Type	Documentation
Event	SHM_Module.Event	Used to get all associated strain_gauge entities.

Return type

List of SHM_Module.strain_gauge

Microflow 'GetStrainGaugeDataSorted'

Returns a list of sorted strain gauge data.

Parameters

Name	Type	Documentation
Event	SHM_Module.Event	Used to get all associated strain_gauge entities.

Return type

List of SHM_Module.strain_gauge

Microflow 'GetUserAssets'

Retrieves a list of the Assets associated to the current user.

Parameters

This microflow has no parameters.

Return type

List of SHM_Module.Asset

Microflow 'OpenEventStartPage'

Opens the event start page.

Parameters

Name	Type	Documentation
EventBatch	SHM_Module.EventBatch	Used to get the associated events.

Return type

Nothing

Microflow 'OpenHomepage'

This is the first microflow which is run as the client enters the web site. It executes a deep link and opens the sign in page whether the deeplink is executed or not.

Parameters

This microflow has no parameters.

Return type

Nothing

Microflow 'RegisterUser_m'

Registers a user.

Parameters

Name	Type	Documentation
Login	SHM_Module.Login	The login object with the login information.

Return type

Nothing

Microflow 'SelectableCharacteristics'

Retrieves a list of selectable characteristics given an event.

Parameters

Name	Type	Documentation
Event	SHM_Module.Event	The event that will be associated to the selected characteristic.
EventBatch	SHM_Module.EventBatch	Used to retrieve the associated Events.

Return type

List of SHM_Module.Characteristic

Microflow 'SelectableDevices'

Retrieves a list of selectable devices given an event.

Parameters

Name	Type	Documentation
Event	SHM_Module.Event	The Event used to get the associated (selectable) devices.

Return type

List of SHM_Module.Device

Microflow 'SelectableServices'

Retrieves a list of selectable services given an event.

Parameters

Name	Type	Documentation
Event	SHM_Module.Event	The Event used to get the associated

		(selectable) services.

Return type

List of SHM_Module.Service

Microflow 'SendDataToServer'

Sends a datastring to the server, parses, commits, and deletes the data.

Parameters

Name	Type	Documentation
data	SHM_Module.ClientData	The data containing the datastring which will be stored at the server.

Return type

Nothing

Microflow 'SetAssetOwner'

Sets the current user as the owner of the given Asset.

Parameters

Name	Type	Documentation
Asset	SHM_Module.Asset	The targeted asset which will have the current user as the owner.

Return type

Nothing

Nanoflows

Name	Return type	Documentation
ACT_SignInUser		
ConnectToCharacteristic		Connects to all devices associated with an event.
ConnectToDevice		Either connects or disconnects to all services associated with a given event depending on whether the

		event IsLogging attribute is true or false. If false, the event starts logging, otherwise, the device is disconnected, and the event stops.
DSS_CreateLoginContext		
DSS_CreateLoginContext_web		
DSS_CreateRegisterContext		
DSS_CreateRegisterContext_web		
GetDeviceServices		
GetEventAssets		
GetEventCharacteristic		
GetEventDevice		
GetEventsByBatch		
GetEventService		
GetServiceCharacteristics		
nfGetEventsReversed		
RegisterUser		
RegisterUser_web		

Nanoflow 'ACT_SignInUser'

Parameters

Name	Type	Documentation
Login	SHM_Module.Login	

Return type

Nothing

Nanoflow 'ConnectToCharacteristic'

Connects to all devices associated with an event.

Parameters

Name	Type	Documentation
EventBatch	SHM_Module.EventBatch	Context

Return type

Nothing

Nanoflow 'ConnectToDevice'

Either connects or disconnects to all services associated with a given event depending on whether the event IsLogging attribute is true or false. If false, the event starts logging, otherwise, the device is disconnected, and the event stops.

Parameters

Name	Type	Documentation
eventBatch	SHM_Module.EventBatch	

Return type

Nothing

Nanoflow 'DSS_CreateLoginContext'

Parameters

This nanoflow has no parameters.

Return type

SHM_Module.Login

Nanoflow 'DSS_CreateLoginContext_web'

Parameters

This nanoflow has no parameters.

Return type

SHM_Module.Login

Nanoflow 'DSS_CreateRegisterContext'

Parameters

This nanoflow has no parameters.

Return type

SHM_Module.Login

Nanoflow 'DSS_CreateRegisterContext_web'

Parameters

This nanoflow has no parameters.

Return type

SHM_Module.Login

Nanoflow 'GetDeviceServices'

Parameters

Name	Type	Documentation
Device	SHM_Module.Device	

Return type

List of SHM_Module.Service

Nanoflow 'GetEventAssets'

Parameters

Name	Type	Documentation
Event	SHM_Module.Event	

Return type

List of SHM_Module.Asset

Nanoflow 'GetEventCharacteristic'

Parameters

Name	Type	Documentation
Event	SHM_Module.Event	

Return type

List of SHM_Module.Characteristic

Nanoflow 'GetEventDevice'

Parameters

Name	Type	Documentation
Event	SHM_Module.Event	

Return type

List of SHM_Module.Device

Nanoflow 'GetEventsByBatch'

Parameters

Name	Type	Documentation
EventBatch	SHM_Module.EventBatch	

Return type

List of SHM_Module.Event

Nanoflow 'GetEventService'

Parameters

Name	Type	Documentation
Event	SHM_Module.Event	

Return type

List of SHM_Module.Service

Nanoflow 'GetServiceCharacteristics'

Parameters

Name	Type	Documentation
Service	SHM_Module.Service	

Return type

List of SHM_Module.Characteristic

Nanoflow 'nfGetEventsReversed'

Parameters

This nanoflow has no parameters.

Return type

List of SHM_Module.EventBatch

Nanoflow 'RegisterUser'

Parameters

Name	Type	Documentation
Login	SHM_Module.Login	

Return type

String

Nanoflow 'RegisterUser_web'

Parameters

Name	Type	Documentation
Login	SHM_Module.Login	

Return type

String

Java actions

Name	Return type	Documentation
computeInverseData		
dataParseCommit		

Java action 'computeInverseData'

Type parameters

This Java action has no type parameters.

Parameters

Name	Type	Documentation
acc_gyro_list	List of	

	SHM_Module.acc_gyro	
mass	Decimal	

Return type

Nothing

Java action 'dataParseCommit'

Type parameters

This Java action has no type parameters.

Parameters

Name	Type	Documentation
dataString	String	

Return type

Nothing

JavaScript actions

Name	Return type	Documentation
BLECharacteristicSubscribe		Subscribes to a characteristic given a UUID and commits data to the server on every notification retrieved from the peripheral device.
BLEDeviceConnect		Pairs with a Bluetooth device with a given UUID.
BLEDeviceDisconnect		Disconnect from all Bluetooth devices.

JavaScript action 'BLECharacteristicSubscribe'

Subscribes to a characteristic given a UUID and commits data to the server on every notification retrieved from the peripheral device.

Type parameters

This JavaScript action has no type parameters.

Parameters

Name	Type	Documentation
characteristicUUID	String	
dataType	String	
eventBatch	SHM_Module.EventBatch	
eventReference	SHM_Module.Event	
serviceUUID	String	

Return type

String

JavaScript action 'BLEDeviceConnect'

Pairs with a Bluetooth device with a given UUID.

Type parameters

This JavaScript action has no type parameters.

Parameters

Name	Type	Documentation
eventBatch	SHM_Module.EventBatch	
primaryServiceUUID	String	

Return type

Nothing

JavaScript action 'BLEDeviceDisconnect'

Disconnect from all Bluetooth devices.

Type parameters

This JavaScript action has no type parameters.

Parameters

Name	Type	Documentation
characteristicUUID	String	
eventBatch	SHM_Module.EventBatch	

serviceUUID	String

Appendix F

Scientific Paper

The following appendix comprises the scientific paper that was written as a result of the master's thesis.

REAL TIME DIGITAL TWIN BASED STRUCTURAL HEALTH MONITORING OF BI-CYCLE FRAMES

P. S. Rølvåg, T. Rølvåg[1], B.Haugen[1], and Zeljko Bozic[1]

[1]NTNU, Faculty of Engineering Science and Technology,7491 Trondheim, Norway

[2]UniZQ, Department of Aeronautical Engineering, 10000 Zagreb, Croatia

Abstract: This paper presents a generic approach for implementation of real time digital twin based structural health monitoring of bicycle frames. The main purpose is to develop and benchmark a framework for identification of structural loads and stresses acting on bicycle frames in real time. The authors use a cloud-based solution to track and store applied bicycle loads, while an edge solution is used to display live loads, accelerations and strain time histories on a smart phone mounted on the handlebars. The digital twin is represented by 6x8 look-up table which enables real time strain calculations in 8 virtual strain gages when random combinations of 6 bicycle loads are acting during bicycle ride. The 6x8 look-up table is precomputed by a unit-load method applied to the bicycle frame modelled in a finite element (FE) program. An analytical inverse method for estimation of dynamic bicycle loads is implemented based on a single IMU sensor. The sensor outputs are processed for noise reduction and singularity removal. Additional strain gauges and temporary accelerometers were mounted on the bicycle for model calibration and validation. The presented approach is described in a general manner and is applicable for undamped bicycle frames.

Keywords: Digital Twin (DT), Structural Health Monitoring (SHM), Reduced Order Model (ROM)

1. Introduction

The structural integrity of bicycle frames is critical in off-road ride and handling. The integrity is dependent on the chosen frame material and design. While the materials are either aluminum or composites for high-end off-road bikes, the design variations and dimensions seem to be almost infinite. Contrary to most platform based automotive solutions, bicycle frames are not optimized by well-known design and load constraints. Fashion driven bike designs also contradict frame standardization and integrity optimization.

Hence, the authors decided to benchmark a generic and low-cost digital twin solution for Structural Health Monitoring (SHM) of bicycle frames. The intention was initially to identify dynamic loads applicable to future frame optimization based on smartphone sensors and apps. Most smartphones have embedded IMUs and free apps sampling angular rates and accelerations in 3D at 100 Hz. Simple tests with an iPhone11 Pro running the SensorLog app [1] proved that 100 Hz sampling of IMU data captured most excitations when riding on various city and off-road paths.

Based on such a simple instrumentation most frame loads can be calculated except the applied input torque from the biker causing chain reaction forces. The smartphone sensors can neither identify the distribution of biker weight on the seat, crank and handlebar during ride and handling. The authors therefore wanted to develop a more scalable IoT framework compatible with low-cost hardware and most sensor types applied in SHM.

To detect the input torque the authors installed a StagesPower crank sensor [2]. This sensor is communicating with Bluetooth to an embedded smartphone app. Unfortunately, the StagesPower app cannot be customized to read data from the smartphone sensors. The authors therefore decided to develop an Arduino hardware solution and a new app based on the Mendix low-code development suite [4]. The Mendix software has an open architecture enabling Bluetooth sensor communication, data processing and dashboard development for data visualization. The Mendix system also embed tools for edge and cloud communication.

To estimate the size, direction, and distribution of dynamic bicycle loads, an inverse method also had to be embedded. The purpose of the inverse method is to predict the distributed inertia loads applied on the bicycle twin model to calculate the distribution of stresses and strains at preselected hot spots. The demand for a 100 Hz sampling frequency did not allow simultaneously finite element analysis of a full finite element model.

The hot spots were identified by a virtual brittle lacquer technique and critical load cases in FEDEM [3]. The selected FEDEM solver supports FMU export and real time calculation of strain and stress time histories. However, due to compatibility and cost considerations, the authors decided to replace a FMU based co-simulation with a precomputed static look-up table. This Reduced Order Model (ROM) digital twin solution is also faster to run on Arduino, easier to implement in Mendix, and fully applicable to the linear behavior of the stiff bicycle frame (no dampers).

Hence, this SHM framework will combine a fast edge (Arduino/smartphone) and a ~~slower but more~~ scalable cloud solution implemented as a back-end service in Mendix. On Arduino, the edge solution is configured to operate at 100 Hz in real time, while the cloud solution collects data from the edge four times per second, or 4 Hz. The edge results are displayed on the Mendix WebApp running on a smartphone mounted on the handlebars.

The real time edge solution shall provide operational decision support to the biker during critical bicycle ride and handling. Safety margins with respect to measured accelerations and calculated structural loads are continuously visualized on a dashboard on the smartphone. The edge solution is also embedding real time stress computations and visualization from both physical and virtual strain gages. These stress results are used to benchmark and validate the digital twin model

The cloud solution must enable more comprehensive off-line structural stress and fatigue life computations. Both physical sensor data and digital twin simulation results are simultaneously displayed on a cloud-based dashboard and smartphone App.

The paper is organized in the Digital Twin asset (bicycle), failure modes, model, theory, validation, results and Inverse methods sections. The Digital Twin method section documents the unit load method applied to generate a Reduced Order Model (ROM) of the physical asset (bike). The theory section addresses the applied FEA formulations, basic fatigue modeling, simulation and post processing tools implemented in the SHM framework. The validation and results section demonstrates the capabilities of the SHM framework with respect to load and stress prediction.

2. Common bicycle failure modes to be detected by the SHM framework

The main purpose of this paper is to present a framework for identification of frame loads and stresses. According to the manufacturer, bicycle frames rarely brake since they are generally conservative dimensioned. Although most bikers want solid and reliable frames, the weight penalty is not appreciated by active or professional riders. Hardrocx is therefore searching for optimal designs offering maximum frame integrity at minimum weight.

The selected bike (physical asset) is a 19” Hardrocx Super Motard M4 provided by Hardrocx. The manufacturer also provided a 3D CAD model, enabling an accurate digital twin model. The bike has a rigid undamped aluminum frame possible to represent and solve by a linear FE model and solver.

The most common failure modes are cracks in frame joints due to manufacturing defects [5]. Fractures may also occur in the middle of a pipe due to under dimensioned tubes in the pursuit of weight saving. However, the most common cause of frame breakage on an aluminum frame is overturning or collision causing stresses above the yield limit.

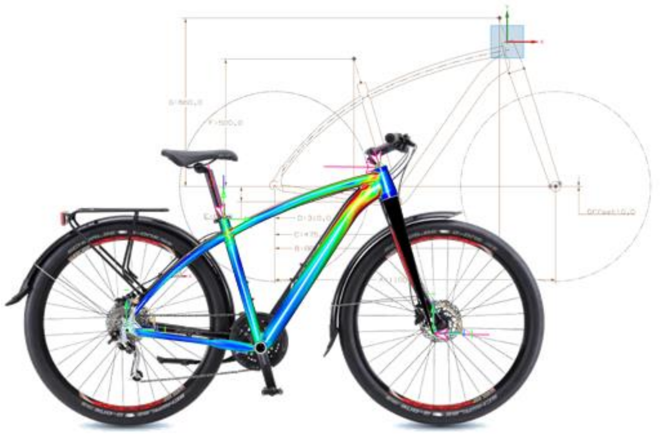


Figure 1 The Hardrocx Super Motard M4

Other cases include seat tube cracks caused by seat pins positioned to high by the customers. If the seat pin does not protrude deep enough into the seat tube on the frame, the frame might break due to flexing back and forth.

The risk associated with these failure modes can be reduced by monitoring the applied frame loads applied in digital twin-based calculations of stress time histories and accumulated damage.

3. The Digital Twin theory

Unit-loads were applied to the FE frame model in FEDEM to precompute the Reduced Order Model (ROM) look-up table (matrix). The ROM can then be multiplied with the estimated load vector in real time to compute gage stresses and strains. FEDEM is also embedded in the cloud solution enabling more comprehensive off-line stress analysis of the frame. The cloud computations will be automated because of the ongoing software integration between SAP (FEDEM) and SIEMENS (Mendix). Although these FEDEM formulations support real time gage stress computations, a static ROM is used since Mendix is not enabling real time co-simulation.

FEDEM is a multidisciplinary simulation system based on a non-linear finite element formulation, CMS model reduction, and control system simulation enabling integrated digital twin modeling and simulation [3,6,7,8]. The nonlinear dynamic FEDEM solver is written on incremental form and solved by the Newmark-β time integration algorithm with respect to the displacement increments $\Delta\mathbf{r}_k$ for time increment k. To achieve equilibrium at the end of the time increment, in the non-linear case, Newton-Raphson iterations must be used to minimize the residual forces:

$$\mathbf{M}_k\Delta\ddot{\mathbf{r}}_k + \mathbf{C}_k\Delta\dot{\mathbf{r}}_k + \mathbf{K}_k\Delta\mathbf{r}_k = \Delta\mathbf{Q}_k \tag{1}$$

where M_k, C_k, and K_k are the system mass, damping and stiffness matrices respectively at the beginning of time increment k. The system mass and stiffness matrices M_k and K_k are Component Mode Synthesis (CMS) reduced, which is the main enabler for real time FE simulation of non-linear systems like complete mountain bikes [6].

$$\mathbf{v}_{free} = \begin{bmatrix}\mathbf{v}_e\\ \mathbf{v}_i\end{bmatrix} = \begin{bmatrix}\mathbf{I} & \mathbf{0}\\ \mathbf{B} & \mathbf{\Phi}\end{bmatrix}\begin{bmatrix}\mathbf{v}_e\\ \mathbf{y}\end{bmatrix} = \mathbf{H}\,\mathbf{v}_{sup} \tag{2}$$

The same technique is basically used to calculate the strain / stress time histories at selected hotspots based on super node displacements as described in [7,**Error! Reference source not found.**].

$$\boldsymbol{\varepsilon}_{rosette} = \left[\mathbf{T}_{re}\widetilde{\mathbf{B}}\ \mathbf{T}\ \mathbf{A}\ \mathbf{L}\ \mathbf{H}\right]\mathbf{v}_{sup} \tag{3}$$

H is the CMS matrix mapping external $\mathbf{v}_{sup}$ to internal displacements $\mathbf{v}_{free}$. The L matrix recover the internal displacements from linear couplings (MPCs). The A matrix extracts nodal displacements defining the strain gage from the full displacement vector. T is transferring the extracted nodal displacements to local strain gage directions. The $\widetilde{\mathbf{B}}$ matrix is the strain-displacement matrix given by the derivatives of the strain element shape functions. The optional $\mathbf{T}_{re}$ matrix transforms the calculated rosette strains and stresses to user defined directions.

The $\left[\mathbf{T}_{re}\widetilde{\mathbf{B}}\ \mathbf{T}\ \mathbf{A}\ \mathbf{L}\ \mathbf{H}\right]$ can be precomputed for each strain gauge element which allows fast real time calculations of hot spot strains during crane operations. This formulation is therefore applicable to digital twin / hardware in the loop applications.

4. The Digital Twin FE model

A physical and a digital 3D model of a 19” Hardrocx SuperMotard M4 bike was received from the manufacturer. The 3D model is an accurate representation of the physical bike. Only minor CAD features not influencing the structural integrity was removed prior to meshing. The bike model was idealized and meshed in NX with 6 mm Teth 10 elements. The FEM model has a total of 154982 elements and 308957 nodes. The tetrahedral elements were assigned specified Aluminum material properties. The seat and steering pin as well as the wheel hubs were represented by RBE2 and RBE3 elements. The meshed model was imported to FEDEM and applied proper boundary conditions in free joints representing the front and rear wheel hubs. Unit loads were later applied at preselected points and directions to generate the Reduced Order Model (ROM)

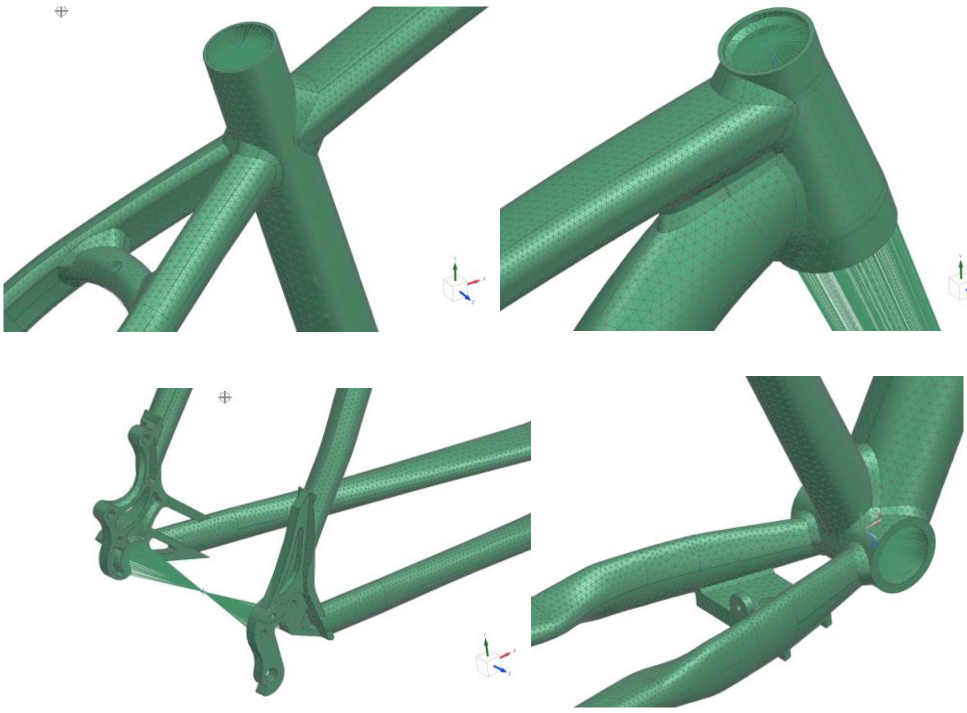


Figure 2 FE model of frame joints

5. The static ROM

FE analysis of the whole frame is too CPU demanding to run real time, and co-simulation is currently not supported by Mendix. Besides, only stresses in hotspots identified by [9] had to be calculated. This trade-off has proven to be an efficient approach since it establishes a linear relationship between a few critical input loads and output hotspot stresses/strains [10]

A static ROM was therefore developed for the 19" Hardrocx bike, by pre-computing the stresses at eight selected frame locations in FEDEM. These locations were highly loaded hotspots previously identified by [5,9]. Single leg strain gages located in the hotspots could then provide stress time histories for fatigue calculations.

Therefore, the static ROM is an 8 x 6 matrix (look-up table) representing the load–stress relationship for the Hardrocx 19" frame. Hence, the six structural loads computed by the inverse method are multiplied by the 8x6 matrix to get the stress distribution in the eight strain gages.

The figure below shows the stress contributions from each applied unit load in FEDEM. These stress distributions are too time-consuming to calculate in real-time, but they were used to find the best strain gage locations. The strain gage results are then calculated in real-time by multiplying the ROM matrix and the load vector estimated by the inverse method.

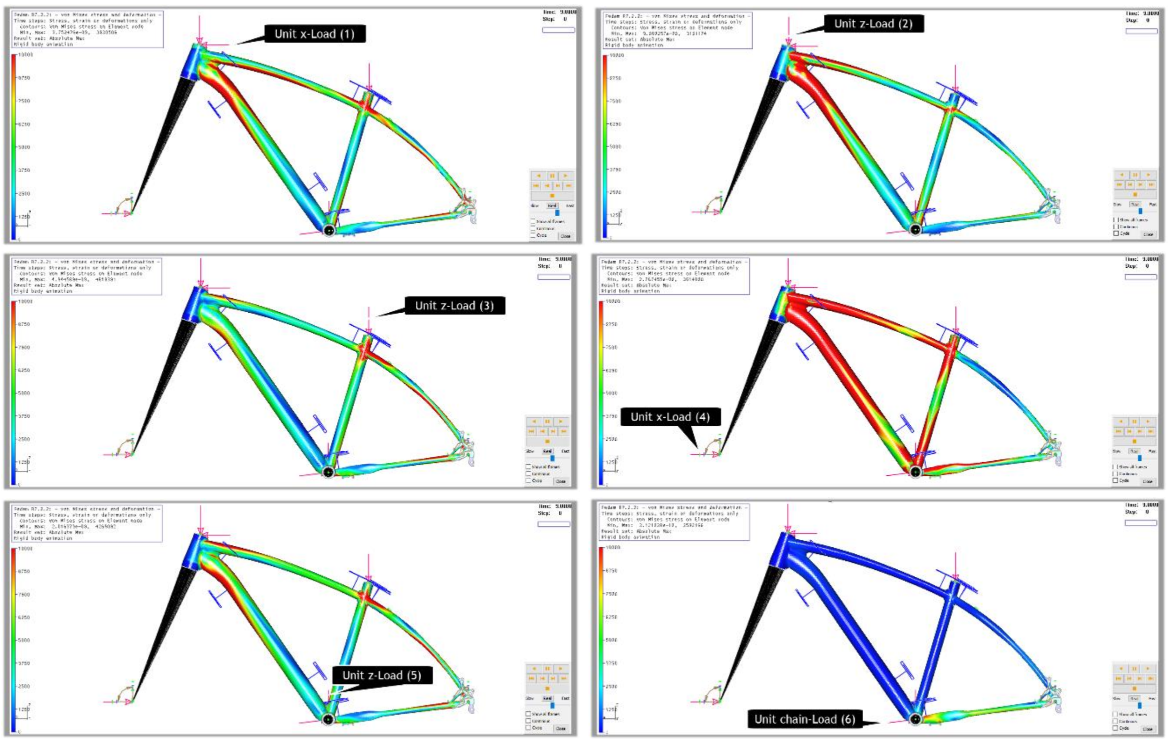


Figure 3 The selected strain gauge locations

The 8 gage results corresponding to the 6 individual unit loads can be assembled in a matrix as shown in Table 1. Since the model is linear and solved for identical boundary conditions, the gage stresses due to any combination of unit loads applied on the FEDEM model or multiplied by the ROM matrix, gives identical results. Hence, any combination of real loads calculated by an inverse method, can be multiplied by the ROM matrix to estimate the real stress time histories. The static ROM is only applicable when the stresses are below yield. Then stress contributions from the various physical loads can be superimposed. Finally, stresses above yield will be captured by a trigger event in the smartphone app.

Table 1: The Unit-Load Matrix calculated by FEDEM for the 19" Hardrocx frame.

Stress [Pa]-in Gage-id	***Unit-Load 1***	***Unit-Load 2***	***Unit-Load 3***	***Unit-Load 4***	***Unit-Load 5***	***Unit-Load 6***
1	2,62E+02	-4,15E+03	-1,25E+03	9,88E+03	-7,31E+02	4,60E+02
2	1,08E+04	3,20E+03	-2,32E+03	-1,87E+04	-1,85E+03	-3,95E+03
3	-1,38E+04	-2,89E+04	-6,02E+03	8,45E+04	-8,59E+03	4,04E+01
4	1,56E+04	2,56E+04	1,18E+04	-7,83E+04	1,64E+04	-9,89E+01
5	1,90E+04	-8,74E+03	-3,03E+04	2,41E+03	-2,05E+04	8,46E+02
6	1,87E+04	-8,82E+03	-3,03E+04	2,86E+03	-2,04E+04	9,35E+02
7	1,51E+04	4,34E+03	-3,47E+03	-2,58E+04	-3,23E+03	-4,76E+03
8	1,02E+04	2,76E+03	-1,98E+03	-1,70E+04	-4,83E+03	-2,26E+02

6. Inverse Method for bicycle ride load prediction

Several methods can be used to calculate bicycle stresses based on estimated bicycle loads. Covill &Al [9] gives a comprehensive overview of methods used to calculate stresses caused by various load cases. They also identified the most critical load cases to be road bumps at the front and rear wheel causing vertical accelerations. Brake forces are less critical but fast to compute and hence included in this study.

The TrueLoads software [10] is a more recent approach used to capture bicycle loads from physical strain gages. The strain gage distribution is optimized based on unit loads applied to defined points and directions on a linear FE model of the physical asset. This method is applied on TREK bikes to capture loads used in frame design and optimization [10]. TrueLoads was the preferred tool in this study, but the hot-spots were already identified in [5,9] as shown in Figure 4. Both physical and virtual strain gages were located on these hot spots for final Digital Twin validation. TrueLoads would also require an expensive DAC and minimum 12 extra strain gages for the identification of the 6 most critical loads shown in Figure 5.

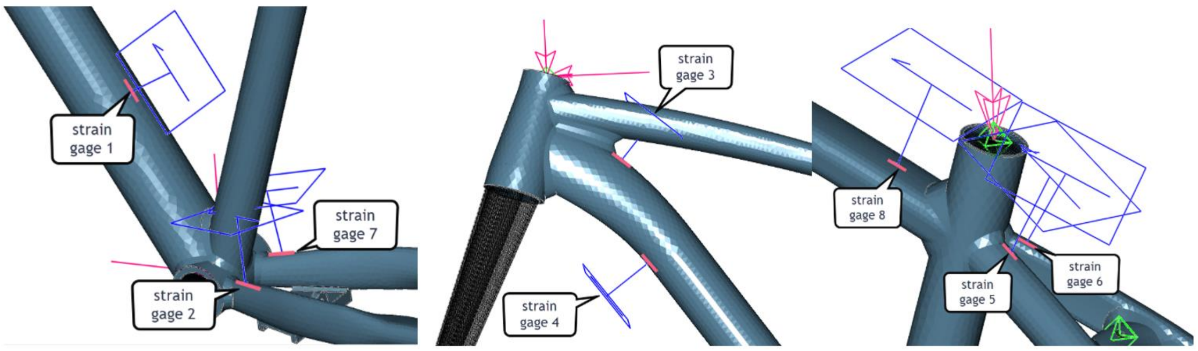


Figure 4 Strain gage distribution

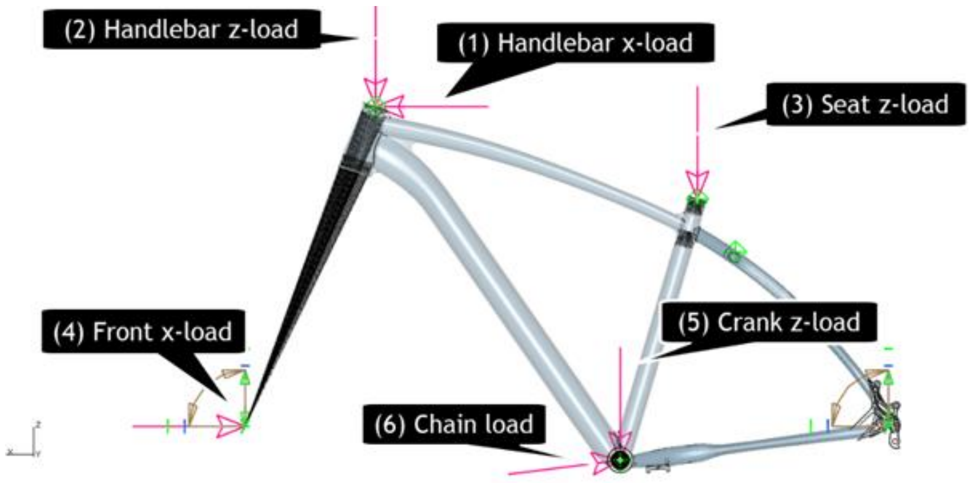


Figure 5 Applied frame loads (vertical hub loads are reaction forces in free joint springs)

The authors therefore decided to estimate the frame inertia biker loads based on outputs from a single low-cost Arduino IMU located on the luggage rack. Based on the measured accelerations (a_x, a_y, a_z) and angular rates ($\dot{\alpha}_x, \dot{\beta}_y, \dot{\gamma}_z$) about the global coordinate system shown in Figure 1, the most critical frame loads during bicycle ride can be estimated in real time as shown in Table 2. The vertical loads acting in the front and rear hub due to acceleration and braking are included as reaction forces in free joint springs. These are non-linear compressions springs acting as tire models.

Since measured vertical acceleration a_z is introduced by both front and rear wheel bumps, a vertical load distribution is calculated based on the bicycle geometry and angular bicycle pitch acceleration $\ddot{\beta}_y$ calculated as the derivative of $\dot{\beta}_y$. When the pitch rate is negative, the front wheel is passing a bump (wheel lift) and vice versa for positive pitch rates and rear wheel bumps.

The estimated rider mass distribution (20% handlebar, 50% seat and 30% crank) is based on physical ride tests on the Hardrocx bike. The weight distribution is obviously depending on the riding position, so all physical tests are performed in the same seated test position. Identification of bicycle loads acting during various seating positions and offroad handling would require additional load cells on the seat and handlebar tubes to capture the rider mass distribution. Such a force driven digital twin setup is more accurate but also contradicts the desired simplicity and current budget constraints.

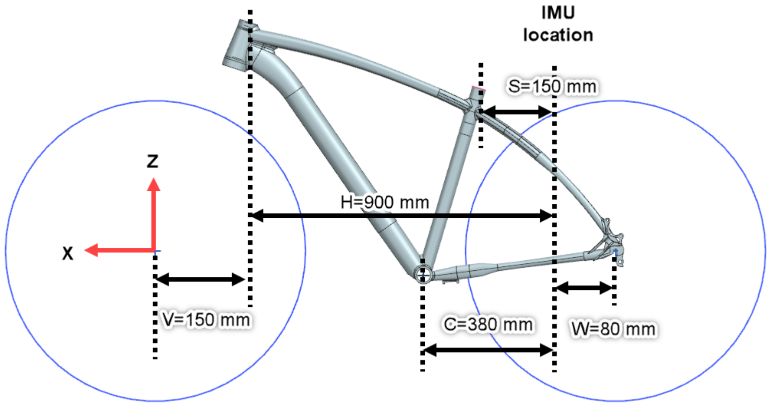


Figure 6 Bicycle geometry used in acceleration calculations

The measured IMU pitch rate $\dot{\beta}_{yIMU}$ and vertical acceleration a_{zIMU}, are used to calculate the accelerations during front and rear wheel bump passes:

Handlebar acceleration	$a_{zH}= H\ddot{\beta}_{yIMU} + a_{zIMU}$	(4)
Crank acceleration	$a_{zC}= C\ddot{\beta}_{yIMU} + a_{zIMU}$	(5)
Seat acceleration	$a_{zS}= S\ddot{\beta}_{yIMU} + a_{zIMU}$	(6)

Where the pitch acceleration is given by $\ddot{\beta}_y = \frac{\dot{\beta}_{y,t+1}-\dot{\beta}_{y,t-1}}{2*dt}$ (dt = sampling time increment). These accelerations are response inputs to an inverse method shown in Table 2 capturing the most important dynamic loads acting during bicycle ride. Inverse methods capturing handling loads will be developed in future implementations, but the current hardware (Arduino shells) and instrumentation (one IMU) is too limited for real time offroad handling loads calculations. However, the real inverse method implementation is more complex and tuned by initial tests.

Table 2 Load calculations / Inverse method

ID	Force	Sensor Output	Property	Inertia Loads by Inverse Method	Description
1	Handlebar x-load	a_{xIMU}	Mass	ma_x	Biker loads due to a_x (mainly weight transfer during breaking) (100%)
2	Handlebar z-load	a_{zIMU}	Mass	$0.2ma_{zH}$	Biker loads due to gravity and a_z (20% weight distribution)
3	Seat z-loads	a_{zIMU}	Mass	$0.5ma_{zS}$	Biker loads due to gravity and a_z (50% weight distribution)
4	Front x-load	a_{xIMU}	Mass	ma_{xIMU} ($a_{xIMU} < 0$) $0{,}02ma_{zIMU}$ ($a_{xIMU} > 0$)	Front brake load (assume 100% of total brake load) or 2% rolling resistance)
5	Crank load z	a_{zIMU}	Mass	$0.3ma_{zC}$	Biker loads due to gravity and a_z (30% weight distribution)
6	Chain load	Torque	Pedal (radius)	Torque/crank radius	Applied compression load from crank torque (in lower arm)

This analytical inverse method for load estimation based on IMU response data, and the static ROM for hot spot stress calculations can both run in real-time with a 100 Hz sampling frequency. However, in practical benchmarks, the sampling frequency dropped to 70Hz when the 8 strain gages and IMU were sampled simultaneously. The inverse method is based on simplified but fast analytical calculations that estimate the main vertical loads and weight transfer during acceleration and braking during bicycle ride. This response driven method may be further developed to capture handling loads. However, light weight force transducers mounted on the handlebar and seat tubes can capture the sprung inertia loads from the biker directly and eliminate the need for an inverse method. The current single IMU based response driven inverse method is fast but limited to straight forward bicycle ride and not general offroad handling. A force driven digital twin is a simpler and more accurate solution but most force transducers are too heavy (see **Error! Reference source not found.**) and will sacrifice the simplicity of the proposed SHM framework.

Digital Twin Validation

The proposed static Reduced Order Model (ROM) shown in Table 1 is only valid if the bicycle frame has a linear behavior. A physical test was therefore conducted to benchmark the physical and digital bicycle model. The seat tube was replaced with a force sensor mounted on an equivalent aluminum tube and 200 kg was applied by a forklift as shown in **Error! Reference source not found.**. The force sensor and stresses in 4 gauges, limited by the available hardware, were recorded simultaneously.

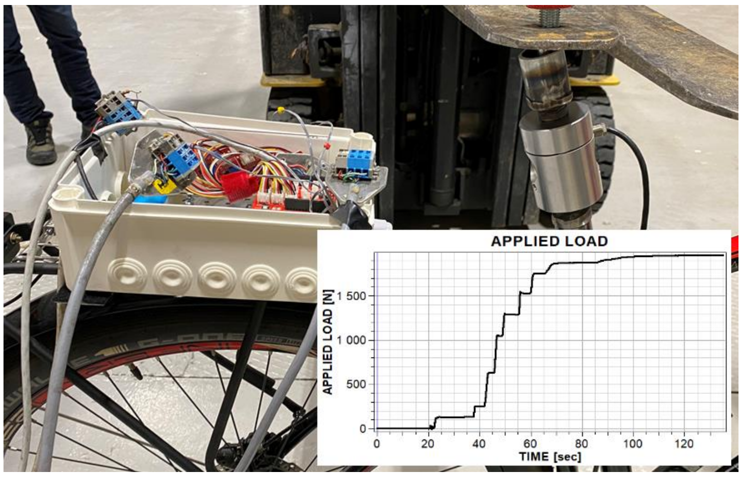


Figure 7 Physical benchmark of Digital Twin

Output stresses from the physical test was recorded and compared with the simulated gage stresses as shown in the left graph in Figure 8. The correlation is very good, and the right graph shows the linear relation between the simulated versus the measured stresses. This proves that the digital twin FEDEM model predicts the real physical stresses and proves that the FEDEM model can be represented by a static ROM due to the linear structural behavior.

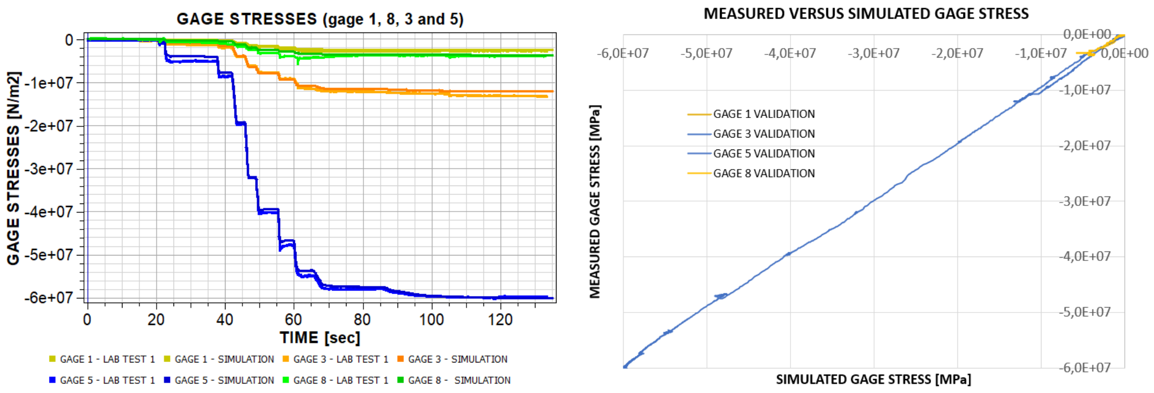


Figure 8 Simulated versus measured gage stresses

7. Digital Twin Test Results

The main objective with this study was to implement and benchmark a generic and low-cost digital twin based SHM system applicable to bicycle frame optimization.

To achieve this, the authors used Arduino hardware a cloud-based solution to track and store applied bicycle loads, and a Mendix app to display live loads, accelerations and strain time histories on a smart phone mounted on the handlebars. These general-purpose tools offered low code software solutions easy to develop and maintain.

However, they also represented performance restrictions due to the applied Bluetooth technology and JavaScript limiting the sampling rate to 100-200 Hz depending on the number of channels. When sampling both the IMU and 8 gage sensors, the sampling rate dropped to 70 Hz.

Initial tests passing the standard speed bump shown in Figure 9 at 20 [km/h], proved that a higher sampling rate was needed to capture the dynamics initiated by the combination of stiff tires and transient impact loads when hitting the bump.

Due to the limited sampling rate when logging all 8 strain gages and IMU sensors, smoother test rides were selected as shown in Figure 10. However, the selected Bromstadekra road has a rough surface and the highest speed bumps in Trondheim, famous for crushing front spoilers!

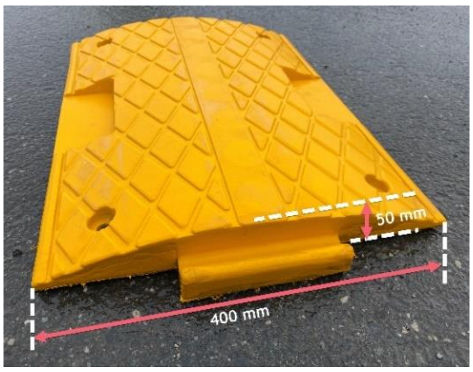
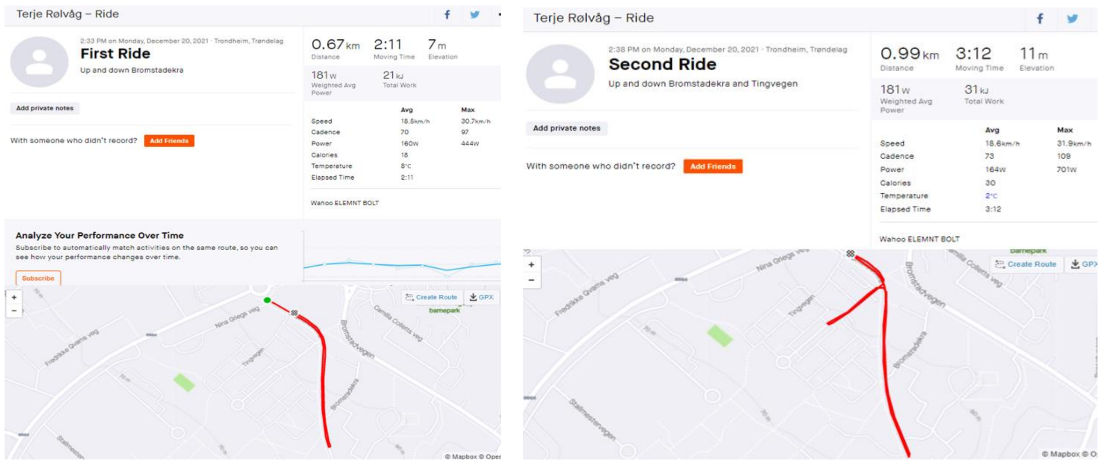


Figure 9 Standard speed bump



Terje Rølvåg – Ride

2:33 PM on Monday, December 20, 2021 · Trondheim, Trøndelag

First Ride

Up and down Bromstadekra

0.67 km Distance | 2:11 Moving Time | 7 m Elevation

181 W Weighted Avg Power | 21 kJ Total Work

	Avg	Max
Speed	18.5km/h	30.7km/h
Cadence	70	97
Power	160W	444W
Calories	18	
Temperature	8°C	
Elapsed Time	2:11	

Wahoo ELEMNT BOLT

Add private notes

With someone who didn't record? Add Friends

Analyze Your Performance Over Time

Subscribe to automatically match activities on the same route, so you can see how your performance changes over time.

Subscribe

Terje Rølvåg – Ride

2:38 PM on Monday, December 20, 2021 · Trondheim, Trøndelag

Second Ride

Up and down Bromstadekra and Tingvegen

0.99 km Distance | 3:12 Moving Time | 11 m Elevation

181 W Weighted Avg Power | 31 kJ Total Work

	Avg	Max
Speed	18.6km/h	31.9km/h
Cadence	73	109
Power	164W	701W
Calories	30	
Temperature	2°C	
Elapsed Time	3:12	

Wahoo ELEMNT BOLT

Add private notes

With someone who didn't record? Add Friends

Figure 10 Selected test rides (Trip 1 left and Trip 2 right)

The IMU sensors recorded vertical a_{zIMU} and longitudinal a_{xIMU} accelerations in the range of 0-2.5G and pitch rates $\dot{\beta}_{yIMU}$ up to 60 rad/sec, used by the inverse method to calculate the handlebar a_{zH}, seat a_{zS} and crank a_{zC}, and gyro pitch $\ddot{\beta}_y$ accelerations. These measurements are sensitive to numerical noise and sample rates and a 10 Hz low pass filter was applied to smoothen the transient accelerations. The measured accelerations are used to calculate the applied inertia loads based on the estimated rider mass. This is a very conservative approach since the IMU is rigidly attached to the bike frame while the rider's body is highly damped and acts like a low pass filter on the transmitted bike accelerations.

In future implementations, the inverse method will be replaced by force transducers fixed to the handlebar and seat tubes to capture the applied inertia loads directly. Force driven digital twins are much more robust than response driven twins influenced by numerical noise from accelerometers or strain gages [6].

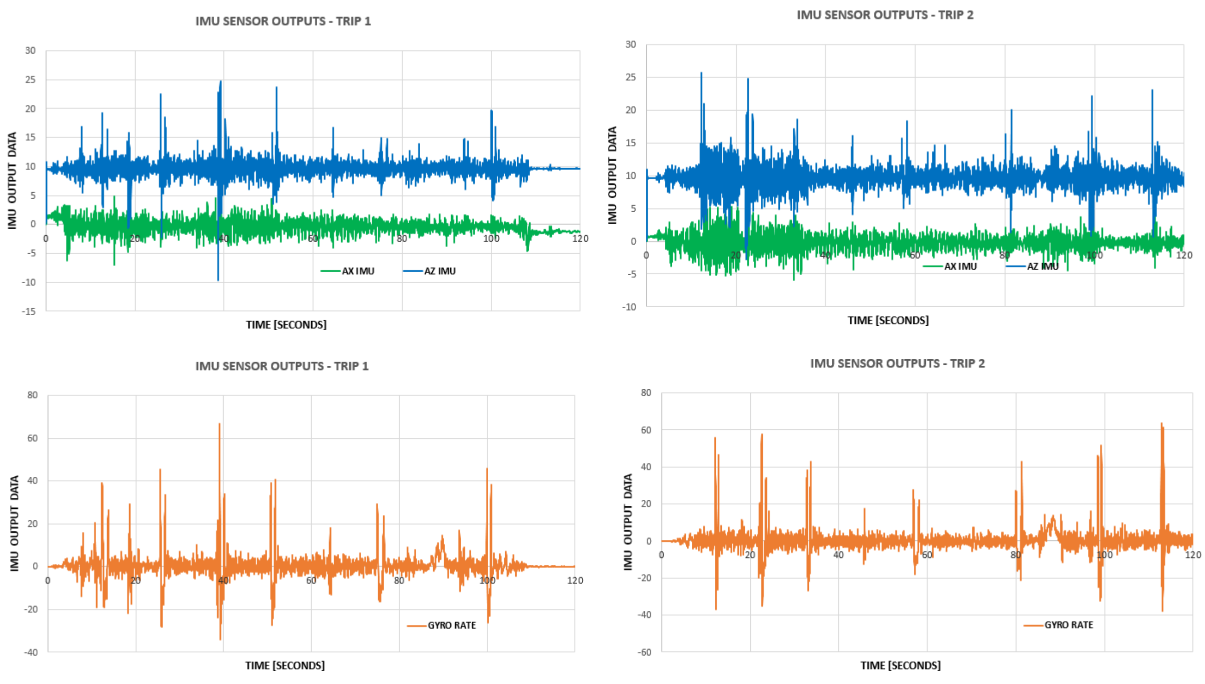


Figure 11 IMU Outputs $(a_{xIMU}, a_{zIMU}$ *and* $\dot{\beta}_{yIMU})$

To estimate the un-sprung riders body mass causing inertia loads, a mannequin is needed that accurately replicate the damping and flexibility of the rider body when excited by the rigid body bicycle motion. This is a complex task which will be addressed in future work. The observed effective un-sprung handlebar, seat and crank rider mass are therefore tuned against the physical strain gage measurements in a simple test passing the speed bump shown in Figure 9.

Based on the physical IMU sensor outputs the rigid body handlebar (a_{zH}), seat (a_{zH}) and crank (a_{zH}) accelerations are calculated by Eq. 4, 5 and 6. These are shown in Figure 12.

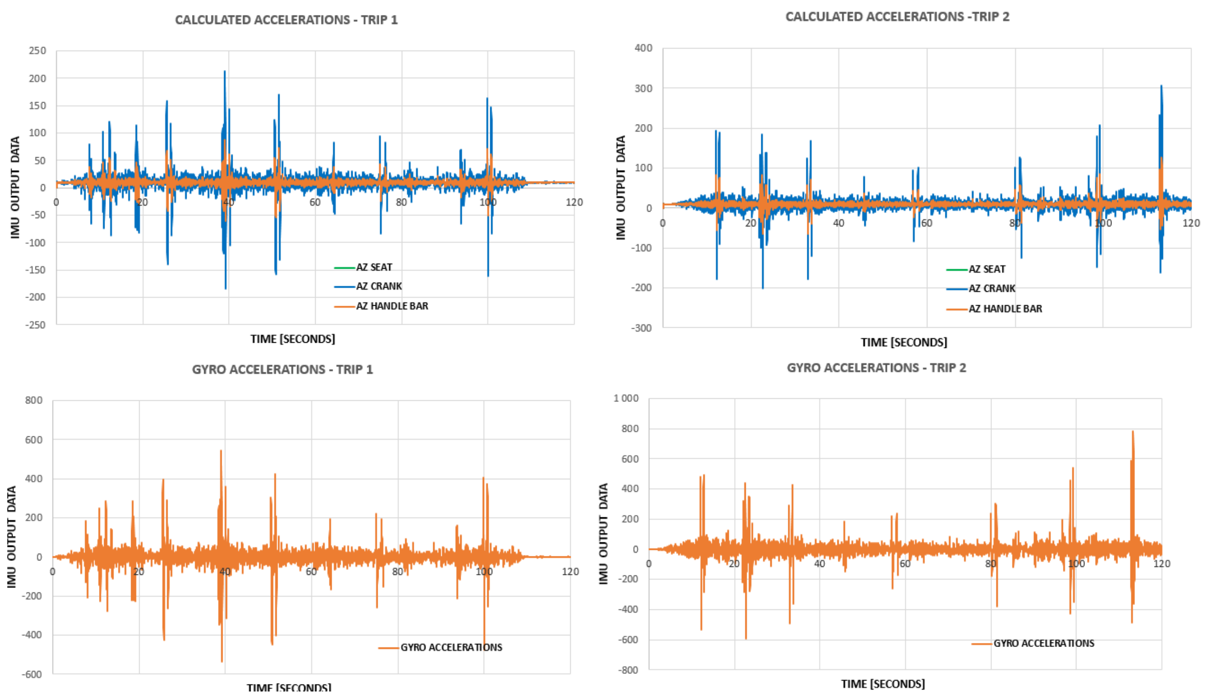


Figure 12 Calculated rigid body accelerations

These computed accelerations are used by the inverse method to calculate Load 1-6 shown in Figure 5.

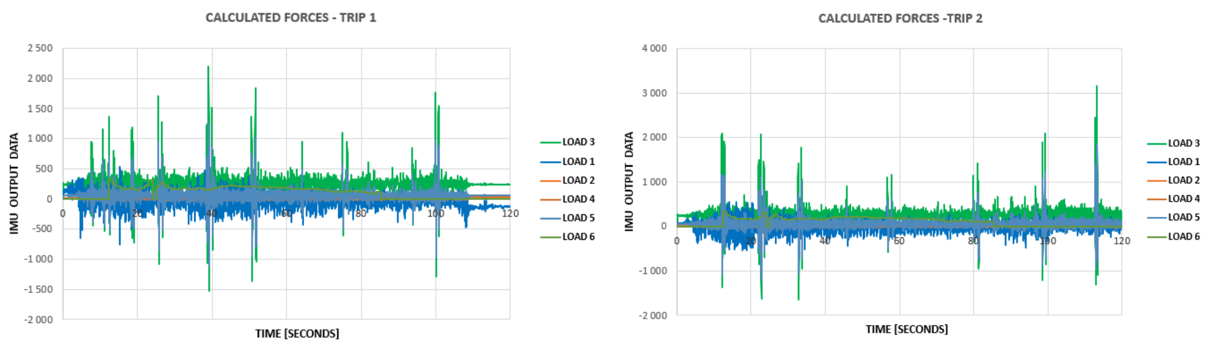


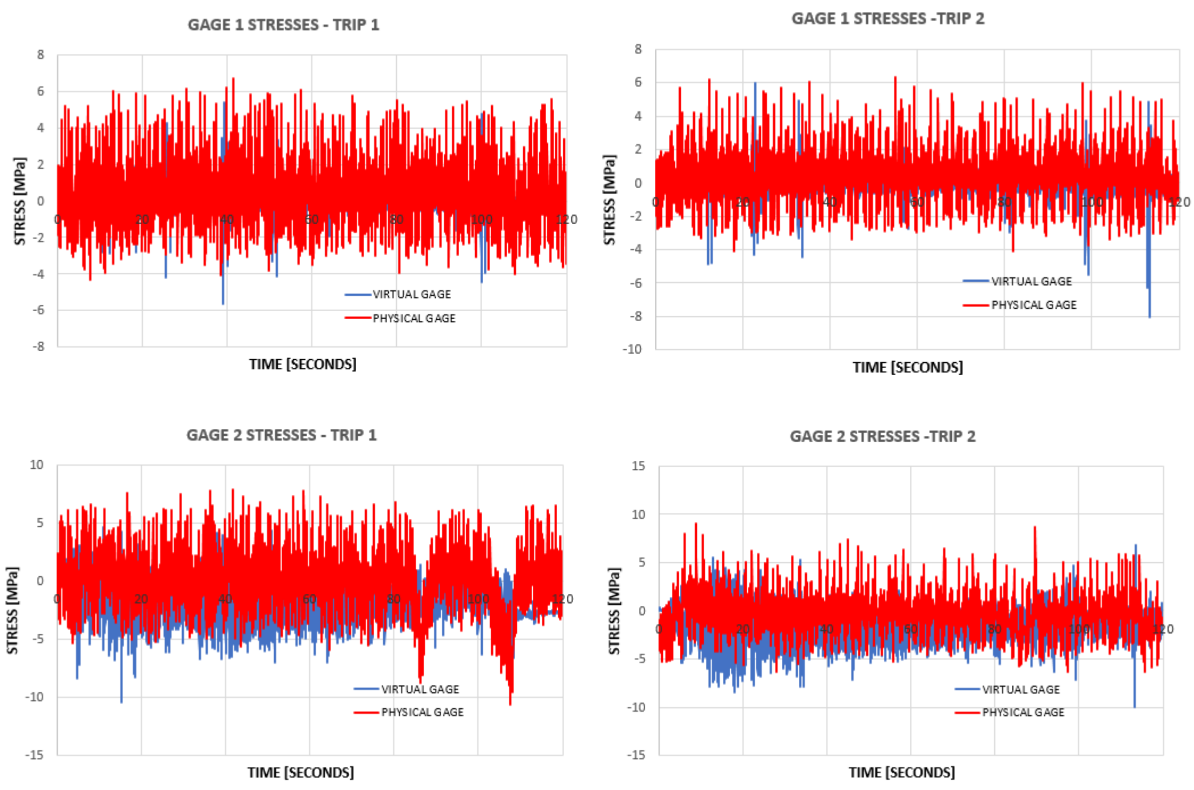
Figure 13 Calculated Ride Loads

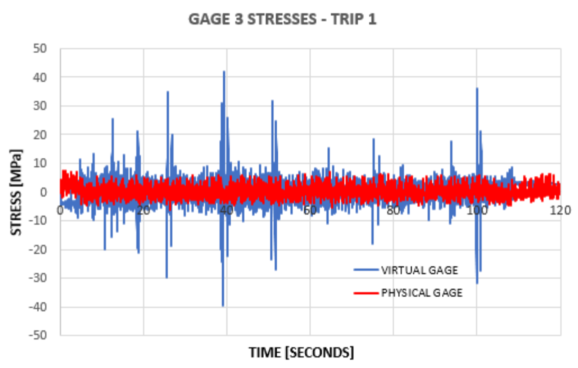
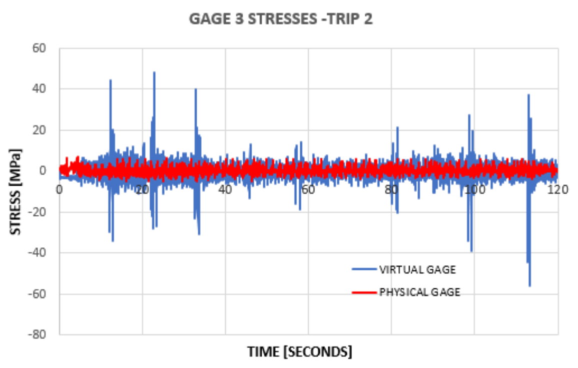
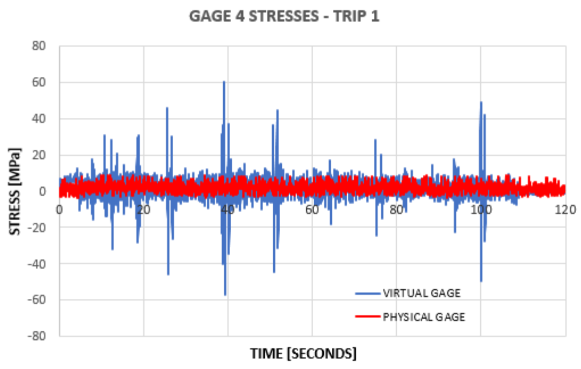
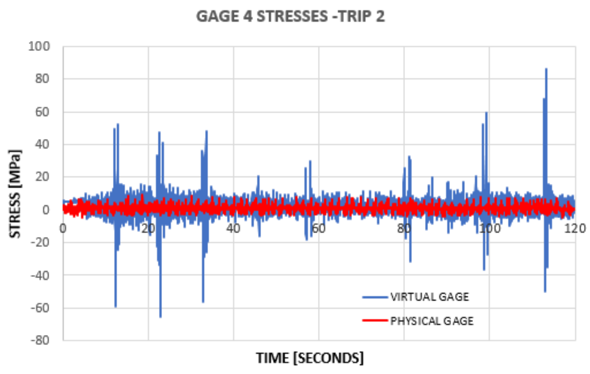
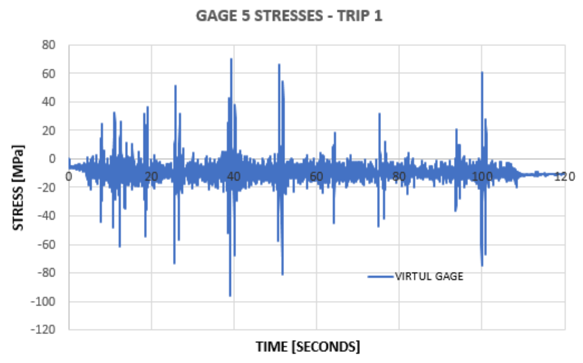
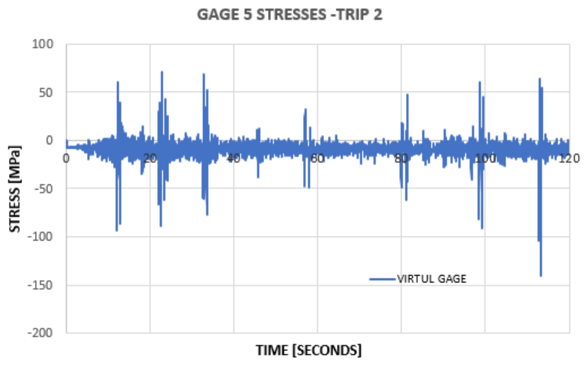
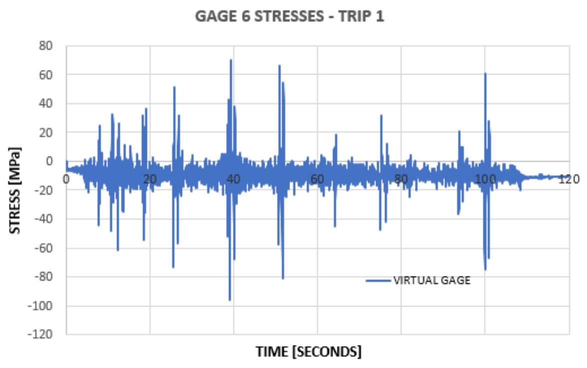
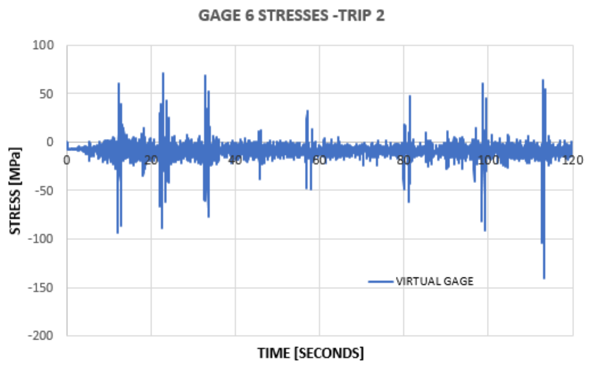
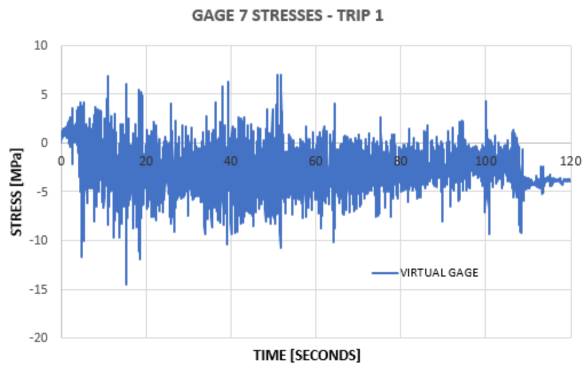
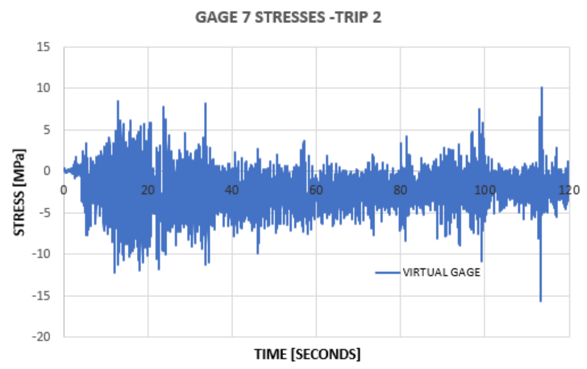
The stresses for the 8 virtual gages are then computed by multiplying the load vector (Load 1-6) with the precomputed FEM based ROM shown in Table 1. The ROM is validated by the physical test as shown in Figure 8. The virtual and physical strain gage stresses are shown in Figure 14.

As seen from the graphs, the stress correlation is good for gage 1 and 2 located on the tubes in the crank region. The stresses in virtual gages 3, 4 and 8 were too conservative compared to the outputs from the physical gages. This is a direct consequence of the IMU accelerations seen in Figure 11 which are representing the rigid body bicycle motion and not the sprung / damped acceleration of the biker's body. In future work, an additional IMU will be located on the biker's body to capture the real biker inertia loads applied to the bicycle frame.

This will also identify the transfer function or dynamic amplification factor of un-sprung bike to sprung/damped biker body accelerations. By tuning the recorded IMU accelerations until match between virtual and physical strain gage stresses the dynamic amplification factors are estimated to be in the range of 0.05-0.1 for the handlebar mass, 0.2-0.4 for the crank mass and 0.5-0.6 for the seat mass.

However, this requires up-front knowledge of future test results and is not an acceptable approach. Physical tests with additional body IMUs will therefore be conducted to identify the sprung body mass and hence more correct inertia forces before a final paper will be submitted to ICSID2022.




GAGE 3 STRESSES - TRIP 1
STRESS [MPa]
TIME [SECONDS]
VIRTUAL GAGE
PHYSICAL GAGE

GAGE 3 STRESSES -TRIP 2
STRESS [MPa]
TIME [SECONDS]
VIRTUAL GAGE
PHYSICAL GAGE

GAGE 4 STRESSES - TRIP 1
STRESS [MPa]
TIME [SECONDS]
VIRTUAL GAGE
PHYSICAL GAGE

GAGE 4 STRESSES -TRIP 2
STRESS [MPa]
TIME [SECONDS]
VIRTUAL GAGE
PHYSICAL GAGE

GAGE 5 STRESSES - TRIP 1
STRESS [MPa]
TIME [SECONDS]
VIRTUL GAGE

GAGE 5 STRESSES -TRIP 2
STRESS [MPa]
TIME [SECONDS]
VIRTUL GAGE

GAGE 6 STRESSES - TRIP 1
STRESS [MPa]
TIME [SECONDS]
VIRTUAL GAGE

GAGE 6 STRESSES -TRIP 2
STRESS [MPa]
TIME [SECONDS]
VIRTUAL GAGE

GAGE 7 STRESSES - TRIP 1
STRESS [MPa]
TIME [SECONDS]
VIRTUAL GAGE

GAGE 7 STRESSES -TRIP 2
STRESS [MPa]
TIME [SECONDS]
VIRTUAL GAGE

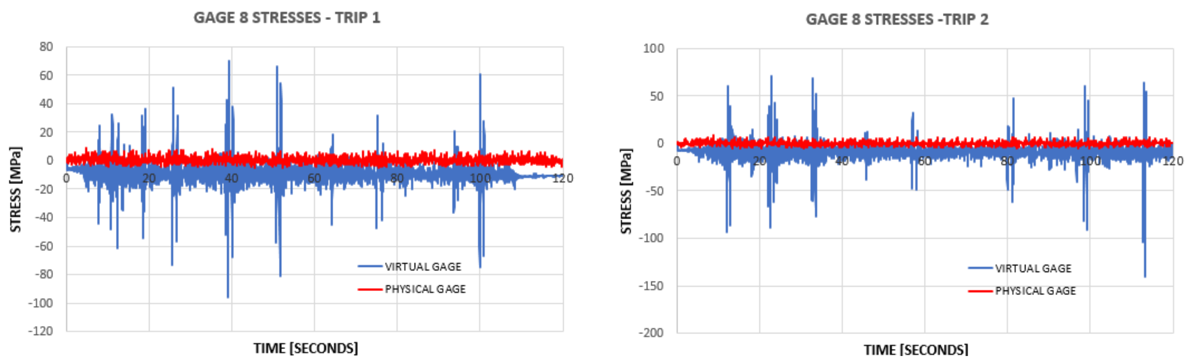


Figure 14 Virtual and physical gage stresses for trip 1 and 2

8. Summary

The main purpose of this paper was to present a framework for real time identification of frame loads and stresses used in in future bicycle designs. Hence, the authors decided to benchmark a simple but fast digital twin solution for Structural Health Monitoring (SHM) of bicycle frames.

Based on a single IMU most frame loads acting during bicycle rides can be calculated except the applied input torque from the biker causing chain reaction forces. The authors therefore wanted to develop a more scalable IoT framework embedding local edge computing and results visualization as well as long term cloud storage.

The authors therefore decided to develop an Arduino hardware solution and a new app based on the Mendix low-code development suite. The Mendix software has an open architecture enabling Bluetooth sensor communication, data processing and dashboard development for data visualization. The Mendix system also embed tools for edge and cloud communication. This framework is scalable but physical tests proved that the expected sampling frequency dropped from 100-200 Hz to 70 Hz in average when both strain gages and IMU data were sampled simultaneously.

To estimate the biker induced inertia loads acting on the biker during ride, an inverse method was developed. The purpose of the inverse method is to predict the distributed dynamic loads applied on the digital twin model to calculate the distribution of stresses and strains at the preselected 8 hot spots. While the bicycle motions are well captured by the single IMU, the sprung motion of the biker's body is hard to validate. The first challenge was to estimate the biker mass distribution on the handlebars, seat and crank without load cells on the frame tubes. The next challenge was to identify the transfer function or dynamic amplification factor of un-sprung bike to sprung/damped biker body accelerations. These two issues will be addressed to better estimate the applied inertia loads before a final paper is submitted. The virtual strain gage stresses are a linear function of the estimated inertia loads and will hence be more correct. The ultimate goal is to eliminate the inverse method by force transducers and hence obtain a robust force driven digital twin with faster performance.

The demand for a 100 Hz sampling frequency did not allow simultaneously finite element analysis of a full finite element model and the unit-load method was applied on a FE model to identify the hot-spots based on a virtual brittle lacquer technique in FEDEM. Then a Reduced Order Model (ROM) influence matrix mapping the unit loads to output stresses in the selected hot-spots was established, verified and approved by a physical test. This Reduced Order Model (ROM) digital twin solution is fast to run on Arduino, easier to implement in Mendix, and fully applicable to the linear behavior of the stiff bicycle frame.

Hence, this SHM framework is combining a fast edge (Arduino/smartphone) and a slower but more scalable cloud solution implemented as a back-end service in Mendix. The edge solution runs real time at 100 Hz on Arduino, while the cloud solution is running at 4 Hz. When both IMU and strain gage sensors are sampled, the bandwidth is reduced to 70 Hz due to Bluetooth and JavaScript limitations. In future implementations all processing will be moved to the cloud where nodes will consume and produce data using an MQTT broker. The broker will be linked to a database that will contain the user's historical data, which the client will be able to access on a smartphone attached on the handlebars. The sampling frequency can then be increased to more than 200 Hz, while also becoming more secure. The app visualization refresh rate will also improve by a large margin.

9. Conclusion

This paper presents a fast and simple digital twin based SHM solution for bicycle frames. The framework is combining edge and cloud solutions that performs well but not as fast as expected due to Bluetooth limitations when both IMU and strain gage sensors are sampled. In future versions the Bluetooth solution will be replaced by local edge processing to support a more scalable framework and faster sampling rates capturing more transient dynamic loads and stresses.

The inverse method is calculating the most critical inertia loads during bicycle rides, but the method has several limitations due to the desired simple and low-budget instrumentation. The rigid body bike accelerations measured by the IMU are too conservative and not representing the damped biker body accelerations. The resulting inertia loads, and gage stresses are therefore very conservative. The sprung motion and hence the biker inertia loads can be better estimated by a mannequin model or additional accelerometers mounted on the biker, but both solutions will increase the complexity and hence reduce the system bandwidth. Both options will be explored in future tests.

Physical tests proved that the digital twin ROM matrix could replace the full bike frame FE model without sacrificing precision while offering real time gage stress calculations. In future framework versions, this ROM matrix will therefore be augmented to capture the influence of bicycle off-road handling loads. This will require force transducers mounted on the frame tubes. Such a load driven digital twin is more robust and accurate than a response driven approach based on IMU or strain gage measurements.

Digital Twin driven SHM is still in its infancy. The authors explored several IoT platforms supposed to deliver the speed and flexibility required to perform real time SHM. Most IoT platforms are designed to support logistics and not high-speed edge solutions needed in real time SHM. The authors will therefore develop a Bluetooth free proprietary SHM solution based on MQTT while keeping the Mendix app for off-line visualization of edge data uploaded to a cloud provider.

10. Acknowledgements

The Authors would like to thank SINTEF OCEAN for providing lab test facilities and valuable inputs to the paper. We also want to thank Hardrocx for providing bicycles and lab technician Halvard Støwer for all help and support. The authors also acknowledge any new sponsor supporting light weight force transducers enabling a force driven digital twin.

References

1. SensorLog App, https://apps.apple.com/us/app/sensorlog/id388014573, 2021
2. StagesPower App, https://stagescycling.com/en_us/product/power-meters, 2021
3. Sivertsen OI (2001) *Virtual testing of mechanical systems theories and techniques*. Advances in Engineering 4. Swets and Zeitlinger B.V., Lisse, The Netherlands
4. MENDIX low-code development suite, https://www.mendix.com/low-code-guide/, 2021
5. Petter S Rølvåg, *Structural Health Monitoring of Mountain Bike*, Project thesis, NTNU, 2021
6. T. Moi, A. Cibicik, T. Rølvåg, *Digital twin based fatigue monitoring of a knuckle boom crane*, published at 3rd International Conference on Structural Integrity and Durability (6 2019).
7. T. Rølvåg, B. Haugen, M. Bella, F. Berto, *Fatigue analysis of high performance race engines*, published at 3rd International Conference on Structural Integrity and Durability (6 2019).
8. T. Rølvåg, M. Bella, *Dynamic test bench for motocross engines*, Adv. Mech. Eng. 9 (2017), https://doi.org/10.1177/1687814017726921 168781401772692
9. Derek Covill, Philippe Allard, Jean-Marc Drouet, Nicholas Emerson, *An Assessment of Bicycle Frame Behaviour under Various Load Conditions Using Numerical Simulations*, 11th conference of the International Sports Engineering Association, ELSEVIER Procedia Engineering 147 (2016) 665 – 670, 2016,
10. Tim Hunter, TrueLoads Trek Bicycles Takes Simulation to the Extreme https://www.wolfstar-tech.com/trek, 2021