Bluetooth Serial Communication in Modern Web Development

Samuel Rautiainen

Haaga-Helia University of Applied Sciences
Degree Programme in Business Technologies
Digital Business Opportunities
Master’s Thesis
2024

Abstract

Author(s)
Samuel Rautiainen

Degree
Master of Business Administration

Report/thesis title
Bluetooth Serial Communication in Modern Web Development

Number of pages and appendix pages
51 + 20

This thesis comprises a Constructive Research study focused on a serial and web communication solution. The work was done for a health technology company. The purpose of the research was to build a proof-of-concept application for a case company and enable their research and development, marketing, and sales. A pivotal aspect of the paper is the documented result of a robust and plausible serial communication architecture for seamless interaction and data flow between diverse components: Bluetooth LE device, a React Native mobile application, and a web application (ReactJS) embedded within a React Native WebView.

The thesis outlines the most crucial theory behind the technical requirements involved and practical instructions to implement a similar solution. The theory is a collection of web and mobile development related material. More of, theory includes the fundamental aspects of Bluetooth and BLE technology, including the architecture, protocols, limitations, and configuration. The technical documentation includes in depth explanation how to create a bi-directional serial communication bridge between a mobile phone and a Bluetooth Low Energy peripheral and between a React Native application and a web application running in a WebView. The documentation of the infrastructure can be consumed in pieces or altogether as one infrastructure.

The paper unveils the success and challenges encountered during the development process, explains how the project was conducted, and portrays the results. Albeit, this project is conducted for a health-tech peripheral, the solution is suitable for any Bluetooth Low Energy capable peripheral. Although as the paper interprets the full-scale implementation of a serial communication between all software entities, a reader can also benefit from smaller portions of the document to implement their own product. The results also include project validation, user acceptance and unit testing, and project management practicalities.

Keywords
Bluetooth Low Energy (BLE), React Native, Constructive Research, Serial.

Tiivistelmä

Tekijä(t) Samuel Rautiainen
Tutkinto Tradenomi (YAMK)
Raportin/Opinnäytetyön nimi Bluetooth Serial Communication in Modern Web Development
Sivu- ja liitesivumäärä 51 + 20

Tämä opinnäytetyö on konstruktiivinen tutkimus, jossa kehitettiin sarjaportin, mobiilin- ja web-sovelluksen välisen tietoliikenteen ratkaisu. Työ tehtiin terveysteknologia-alan yritykselle. Työn tarkoitus oli rakentaa yritykselle sovellus, joka mahdollistaa yrityksen aloittaa tutkimaan, markkinoimaan ja myymään yrityksensä palveluita. Tutkimuksen ja tuotekehityksen tärkein aspekti on dokumentoitu vankka sovellusarkkitehtuuri, joka mahdollistaa perusteellisen ja saumattoman kaksisuuntaisen tiedonsiirron Bluetooth LE laitteen virtuaalisen sarjaportin, React Native mobiilisovelluksen ja web-applikaation (ReactJS) välille.

Dokumentti myös kertoo, miten projekti toteutettiin käytännössä, sekä käy läpi projektin lopputuloksen. Projektin käytännön lisäksi esitetään teoriaa kaikista kriittisimmistä aiheista (Bluetooth LE, mobiilituotekehitys, web-kehitys). Myös ratkaisua selitetään käytännönläheisin esimerkein ja raportti sisältää koodiesimerkkejä. Tekninen dokumentointi esittää käytännönläheisesti, miten rakentaa kaksisuuntainen seriaali kommunikointiyhteys Bluetooth LE ja React Native mobiilisovelluksen välille, sekä React Native mobiilisovelluksen ja WebView:ssa olevan web-sovelluksen välille.

Tämä dokumentti esittää myös projektin onnistumiset ja haasteet, jota kirjoittaja kohtasi projektin aikana. Vaikka projekti on tehty terveysteknologia-alalle tarkoitetulle laitteelle, on esitetty ratkaisu täysin sovellettavissa mille tahansa laitteelle, jolla on Bluetooth LE ominaisuuksia. Täten tutkimus on myös sovellettavissa mille tahansa toiminta-alueelle, sillä ratkaisu ei ole sidottu yhteen alaan. Tutkimus sisältää laajamittaisen kattauksen eri teknologioita, mutta lukijan saa hyötyä tutkimalla vain osaa ratkaisusta hyödyntäen yksittäisiä osioita omassa tuotteessaan. Dokumentti sisältää näiden kaikkien lisäksi myös tutkimuksen validointia, yksikkö- ja hyväksyntätestauksen ja projektinhallinnan käytäntöä.

Asiasanat
Bluetooth Low Energy (BLE), React Native, Konstruktiivinen tutkimus, Sarjaliikenne.

Table of contents

1 Introduction

Fepod is a startup company founded in Finland early 2022. Their business domain is health-technology, and their business is about developing, selling, and out-licensing rights for an electrochemical point-of-care blood diagnostics platform.

Measuring blood concentration of paracetamol, opioids, and other pain medicine directly from a drop of blood at the point of care is faster than taking the samples to a laboratory. With Fepod's solution it is possible to get results within seconds on site even when no laboratory is available. It is ideal for first responders working at the field with a mobile phone, an electrochemical potentiostat, and sampling sensors.

Thesis' purpose was to study, research, and build a product, which allows Fepod to begin their business initiatives. The research results into a proof-of-concept product which allows the company to conduct research and development work, marketing, and sales.

In this master's thesis challenges are explained and important questions are answered. The research questions guide the research and maintain focus on the important aspects of the research. The objective is to gather known information in information technology and combine it with new insight gathered during the research process.

There are three research questions related to this thesis:

1. What are the requirements for enabling Fepod to start their business?
2. How to solve the technical problems to fulfill the requirements?
3. How can the solution be adapted to other businesses?

Each chapter focuses on answering each question to fulfill the purpose of the research. The result is to provide an explanation how this research fulfilled the company's business requirements and provide a feasible solution for anyone to implement a similar Bluetooth Serial communication architecture for their business.

The following chapters explain why and how the project was built, the project's success and challenges. Thesis' Theory-chapter dives into the fundamentals about the methologies and technical details about Bluetooth, and both mobile and web development. The Solution-chapter (chapter 4) explains technical detail how to build a serial communication architecture between a Bluetooth Low Energy device, a React Native-application running a web-application in a WebView in ReactJS. The solution includes code examples how to achieve the bi-directional serial and data communication between all entities.

My named role was frontend developer, and my responsibilities were to develop the React Native and ReactJS. The work included understanding Bluetooth technology and serial communication. I was also a part in a team of three, however I was solely working with frontend technologies. In addition to technical responsibilities, I acted as a Scrum-team member.

Figure 1. A Bluetooth LE enabled peripheral device (PalmSens 2023)

To be able to make measurement, a mobile device or a desktop computer must be connected to the measurement device, also known as a peripheral. The peripheral in this case is an electrochemical potentiostat (Figure 1). With potential, current, and voltage, it is possible to determine different substances in liquid, such as blood (Adams, Doeven, Quayle & Kouzani 2019, 31904.)

Bluetooth SIG mentions (Bluetooth SIG 2023f) that in the year 2027 7.6bn devices will be shipped annually with a 9% growth in 5 years. This means new products and users are growing linearly also. It is worth emphasizing that even when new devices are shipped, old devices and services also exist. What it means regarding this serial communication architecture, old web-based services can be upgraded with low code and low effort to support wireless serial communication. In addition, web-based services functioning on a desktop-computer can be enabled to work on Android and IOS devices. These products include IoT devices, health-tech devices, smart home, assistant living for the elderly, medical equipment, any biometric sensors, glucose monitors, activity and motion sensors, industrial machinery, agriculture machinery and sensors, retail, stock and warehouse monitoring, and security. The serial communication architecture can be implemented to old and new devices regardless of the device’s scripting, functionality, or manufacturer if it is based on the standardized Bluetooth protocols.

Although the solution is built for a specific Bluetooth device and business sector, this thesis is adaptable for any Bluetooth Low Energy enabled device and in any business that requires Bluetooth communication, virtual serial ports and/or communication between React Native and web applications via a WebView.

2 Theory

The following chapters consist of information about the key aspects of Bluetooth, Bluetooth Low Energy, Bluetooth profiles, protocols, vocabulary, and associated topics. Also, necessary web and native development tools are portrayed. The Bluetooth SIG documentation is a wide collection of in-depth documentation. This thesis includes the most necessary and relevant topics to create the architecture successfully.

The technical information is selected based on the developer’s experience building the application or software. Bluetooth is a complicated technical establishment. It contains relevant and irrelevant theory for a developer. This theory emphasizes on the topics developer should know or is good to know before implementing the Bluetooth functionality into an application. That said, all these concepts were present and relevant while building the project.

2.1 Bluetooth

In 1996, three main companies in the technology industry, Nokia, Intel, and Ericsson, decided to work together to plan and build a standardized short-range radio technology. The goal was to support connecting devices and collaborate between different products and industries. (Bluetooth SIG 2023d.)

Bluetooth was built for two devices to exchange information between each other without external network equipment. Such devices are Bluetooth keyboards, mice, and headphones. The very first version of Bluetooth is known as Bluetooth BR. BR stands for “Basic Rate” and it supported a raw data rate of 1 mb/s. The next version is called Bluetooth BR/EDR (Enhanced Data Rate) and it doubled the bitrate to 2mb/s. It was Bluetooth version 4.0, also known as Bluetooth Low Energy, Bluetooth LE or BLE, that enabled more possibilities to connect with and broadcast to unlimited receivers simultaneously. Also mesh networking became possible. It allows tens of thousands of devices to communicate with each other instead of one-to-one communication. (Bluetooth SIG 2023b, 7.)

The Bluetooth Interest Group, SIG in short, is a non-profit membership organization that oversees, documents, and manages Bluetooth technology nowadays. The organization is also the owner of the patents related to the technology and research. Bluetooth SIG is a group of companies and members collaborating throughout the entire technology to create new specifications and maintain existing ones. It is the single source of truth when it comes to Bluetooth documentation. Bluetooth SIG suggests that 5.4 billion devices that use or relate to Bluetooth technology will be shipped during the year 2023. (Bluetooth SIG 2023e.)

2.2 Bluetooth Low Energy

The Bluetooth 4.0+ versions are known as Bluetooth Low Energy (also known as BLE or Bluetooth LE). Bluetooth is a wireless technology for data transfer. As its name suggests, compared to its former and older version Bluetooth Classic, BLE works in a way that it consumes less energy. Bluetooth 4.0 was released in 2010. It was created to better support IoT (Internet of Things) devices (Afanef April 2018, min. 1-2). BLE radio transmits data over 40 channels in 2.4GHz frequency (Bluetooth SIG, 2023a).

Figure 2 shows the Bluetooth LE stack in its core. The layers and distribution are visible across the host and controller components. The Host Controller Interface is portrayed between the Host and Controller, as it acts as a logical interface between the two. It is not a physical component, board, or device as such. (Bluetooth SIG 2023b, 10.)

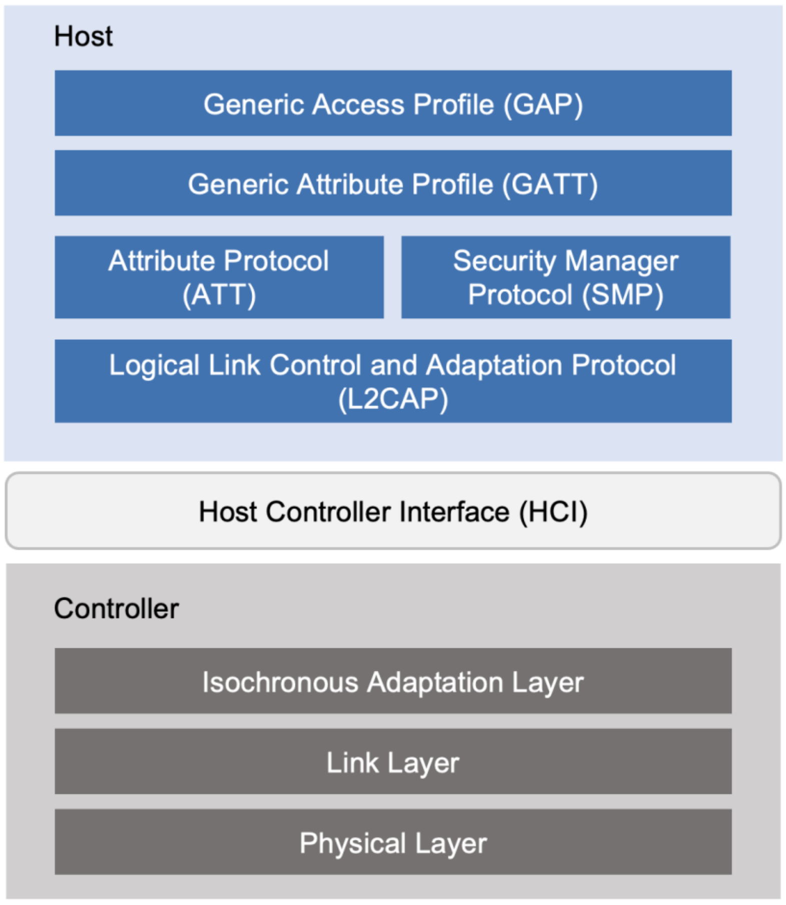
Host
Generic Access Profile (GAP)
Generic Attribute Profile (GATT)
Attribute Protocol (ATT)
Security Manager Protocol (SMP)
Logical Link Control and Adaptation Protocol (L2CAP)
Host Controller Interface (HCI)
Controller
Isochronous Adaptation Layer
Link Layer
Physical Layer

Figure 2. Bluetooth LE protocol stack (adaptive from Bluetooth SIG 2023b, 10)

2.2.1 Frequency Band

Bluetooth LE operates in the 2.5GHz band. Its range is therefore between 2400MHz-2483.5MHz. The frequency is divided into 40 channels with a spacing of 2MHz. The usage of channels is specified by the link layer and data transport architecture. (Bluetooth SIG 2023b, 13.)

Table 1. 40 Bluetooth LE channels with the spacing of 2MHz frequency and the three advertising channels (adaptive from Bluetooth SIG 2023b, 13; Bluetooth SIG 2023c)

RF Channel Frequency (MHz)	Index	Advertising
2402	37	Yes
2404	0	
2406	1	
…	…	
2424	10	
	…	
2426	38	Yes
2428	11	
…	…	
2480	39	Yes

As stated, Bluetooth divides the 2.4GHz frequency into 40 channels. The Link Layer is the main entity that controls how the channels are utilized. Bluetooth LE uses spread spectrum techniques in many ways and transfers data through multiple of these available channels. This is done to prevent collision with multiple simultaneous data transfers and therefore become more stable and reliable. A well-known example of it is adaptive frequency hopping. Adaptive frequency hopping works by changing channels in regular intervals. The system tracks the channels used and evaluates the quality of the passed data. If the transmission quality is weak due to interference or another culprit, the system can set the channel as "unused" to ensure the channel is not used. This way the algorithm choosing the channels uses the best available options for data transfer. (Bluetooth SIG 2023b, 19.)

2.3 Bluetooth Profiles

To accomplish Bluetooth devices built by different manufacturers to work together, there are several Bluetooth profiles in place. The profiles specify the required and necessary functions and features of each layer in the Bluetooth architecture. The profiles handle and define the interactions between layers for a device and peer-to-peer interactions between several devices. Profiles also define application behavior and data formats. Once two devices comply with the requirements set by the profiles, the devices function together. The profiles are crucial to scan and connect the devices and to use device features. (Core Specification Working Group 2023, 278.)

2.3.1 Generic Access Profile (GAP)

The Generic Access Profile (GAP) layer is responsible for connection functionality. This layer handles device discovery, link establishment and termination, security features, device and service discovery, and device configuration. (Texas Instruments 2016a.)

The GAP profile is the base profile for a Bluetooth device. This profile includes the basic features of a device. For Bluetooth LE, the GAP profile defines the physical layer, Link Layer, L2CAP, Security Manager, Attribute Protocol, and GAP. This ties all layers together to make a basic requirement for a device. (Core Specification Working Group 2023, 279.)

There are multiple advertising and scanning packet types which are described by the link layer. Even when advertising and connections are established by the GAP, the procedures are performed by the link layer (more detail about the link layer in chapter “2.7 The Link Layer”). (Bluetooth SIG 2023. 71.)

GAP defines four device roles in Bluetooth LE. “Broadcaster” role is when a device uses advertising to transmit data. Broadcasting does not necessarily need to have an established connection with a device as a broadcaster device does not support any connectivity. A typical broadcasting device is a Beacon. A Beacon transmits data continuously to anyone listening, even if no receiver device is available. An “Observer” receives advertising or broadcast packages without connecting or sending data to other devices. These devices analyze signal quality or monitor security. A “Peripheral” can connect to devices and contains a transmitter and a receiver. A peripheral is typically a Bluetooth device without a virtual user interface. Common peripherals are audio devices, such as headphones, printers, and keyboards. A “Central” can establish a connection to a peripheral and has a transmitter and a receiver. A central typically is a mobile phone or computer connecting to a Bluetooth device and can manage multiple connections simultaneously. A device may support multiple roles and combinations of roles. The controller is by its role defined as how to be optimized to its defined use cases. (Bluetooth SIG 2023. 71; Core Specification Working Group 2023, 279.)

As all Bluetooth devices are required to implement GAP, every other Bluetooth profile is a superset of GAP. This means all other profiles are dependent on GAP (Specification Working Group 2023, 279).

2.3.2 Attribute Protocol (ATT)

To enable devices to read and write data, an Attribute Protocol (ATT) is defined. The attribute protocol is used by two devices. One acts as a client and the other as a server. The server is responsible for sending information about its services. These are known as attributes. Bluetooth devices have properties set and these properties/attributes can be read from an indexed list called an attribute table. These attributes can be viewed when paired with a peripheral and are identified with UUIDs. (Bluetooth SIG 2023b, 61.)

The Attribute Protocol defines two roles: ATT Client and ATT Server. A single device can be both simultaneously. The messages are handled by a single bearer one message at a time in both ways. A single device can have multiple bearers, which allows multiple message transactions to be done concurrently. (Specification Working Group 2023, 280.)

2.4 L2CAP

L2CAP is an abbreviation for Logical Link Control and Adaptation Layer Protocol. The L2CAP layer is positioned on top of the HCI layer on the host side (see Figure 2). Its duty is to transfer data to upper layers of the host, such as GAP and GATT, and to lower layers of the protocol stack. The L2CAP layer is responsible for multiplexing capabilities, segmentation, and reassembly operation for data exchanged between the host and the protocols. (Texas Instruments 2016c.)

Multiplexing means the L2CAP allows multiple protocols to share a single Bluetooth connection. Segmentation and reassembly allow large data packets to be transmitted over the Bluetooth link and reassemble at the receiver (Specification Working Group 2023, 189-190). This is useful when dealing with large volumes of data and is relevant for Bluetooth LE and the topic is related to the device’s MTU.

2.5 Maximum Transmission Unit (MTU)

Due to L2CAP and its definitions, Bluetooth LE supports data fragmentation/multiplexing and recombination to split large data into pieces and recombine it on the receiving end to reduce overhead. The Maximum Transmission Unit (MTU) is a value specified by the receiver device and it tells the sender device the maximum size of a data chunk the receiver can receive. Typically, by default, the MTU of a BLE packet is 23 bytes. The data itself (L2CAP PDU (Protocol Data Unit)) is 27 bytes, but the L2CAP protocol header is 4 bytes, which subtracts the actual Attribute Protocol Maximum Transmission packet size or “ATT_MTU” to be 23 bytes. (Texas Instruments 2016c.)

The MTU is not a negotiated value. It is information set by the peripheral device itself. In other words, the manufacturer defines their device’s MTU. If a device tries to send larger chunks of data than specified in the MTU, the receiver will reject it. (Core Specification Working Group 2023, 1069-1070.)

2.6 Generic Attribute Profile (GATT)

The abbreviation “GATT” stands for Generic Attribute Profile, and it contains higher-level constructs such as services, characteristics, and descriptors of a device. It is a general specification for sending and receiving “attributes” over a BLE link. The characteristics provide an interface to use the device’s properties. The properties include information on the Bluetooth peripheral device’s availability, searchability, connection, readability, or writability. (Afanef June 2018, min. 4-5; Android Developers 2023; Bluetooth SIG 2023b, 61).

Often services correspond to a feature or capability of a device, such as read, write, notify, and connect. Characteristics are individual items of the state of the device and have type and value pairs. All characteristics belong to a service. (Bluetooth SIG 2023b, 67.)

A characteristic contains a single value and 0-n description, which describes the characteristic value. They relate to some internal state of the device or a device’s environment i.e., using the device’s sensor. The nature of the peripheral device determines what characteristics it may have. A heart rate monitor could have a heart rate characteristic and a battery level status, whilst a Bluetooth thermometer’s characteristics would be to measure the temperature and humidity with a sensor. A characteristic can also represent configuration data about the peripheral. For example, a characteristic type and descriptor can determine the frequency of a successful measurement. The characteristic information can be read once connected to a peripheral with a central device. (Android Developers 2023; Woolley 10 August 2016.)

Descriptors are descriptions of the characteristic value. In short, they are human-readable metadata that describe what the characteristics are for. Descriptors may also portray accepted values about a characteristic, a range, unit, or UoM (Unit of Measure). (Android Developers 2023; Woolley 10 August 2016.)

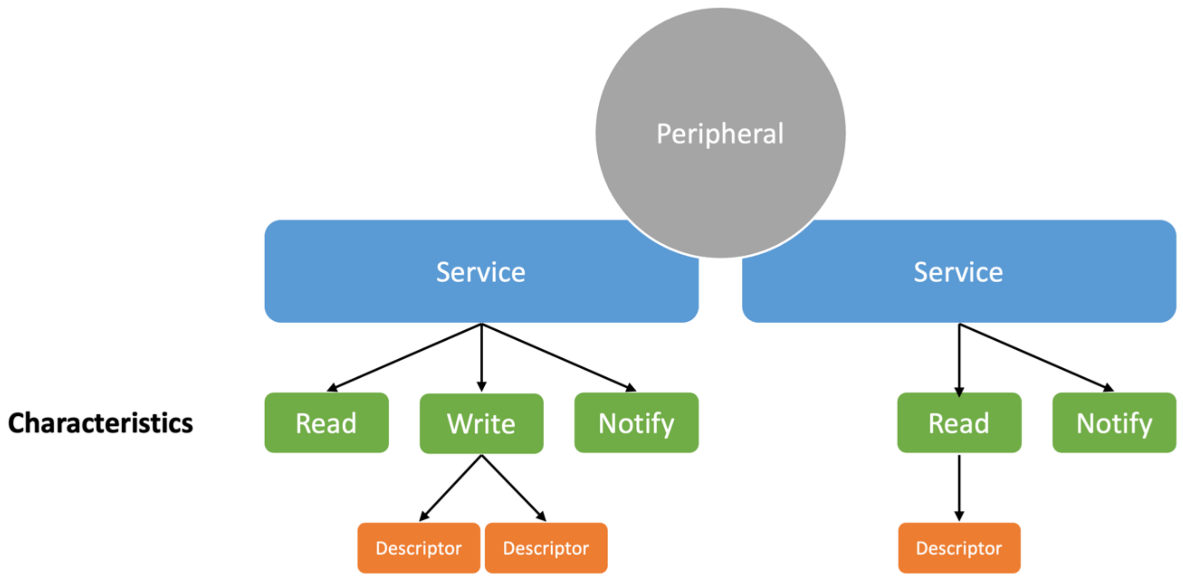

Figure 3. The hierarchy of a peripheral's services, characteristics, and descriptors (adaptive from Bluetooth SIG 2023b, 67)

Two services are mandatory in all GATT servers: the generic access service and the generic attribute service. The peripheral device manufacturer can define custom services, characteristics, and descriptors alongside Bluetooth SIG-specified attributes. They can be identified as 128-bit UUID values or purchase 16-bit UUID values from the Bluetooth SIG. (Bluetooth SIG 2023b, 68.)

2.7 The Link Layer

Bluetooth SIG (Bluetooth SIG 2023b, 16) describes the Link Layer to be one of the largest and the most complicated sections of their BLE specification documentation. The Link Layer has multiple responsibilities across the Bluetooth system. The Link Layer is primarily responsible for managing the physical connections and providing data links between one-to-one and one-to-many connections.

The Link Layer contains the following states:

- Standby
- Advertising
- Scanning
- Initiating
- Connection

- Synchronization
- Isochronous Broadcasting

A device has either the Advertising (peripheral) or Scanning (central) state. In the advertising state, the Link Layer will transmit physical channel packets and listen and respond to packets. A device accesses the advertising state from the standby state. (Bluetooth SIG 2023c.)

The Link Layer in the scanning state listens to the advertising physical channel packets sent by the advertising devices. This scanning device is known as the scanner and the scanning state is entered also from the standby state. When a connection is created between two or more devices, the device enters the connection state. (Bluetooth SIG 2023c.)

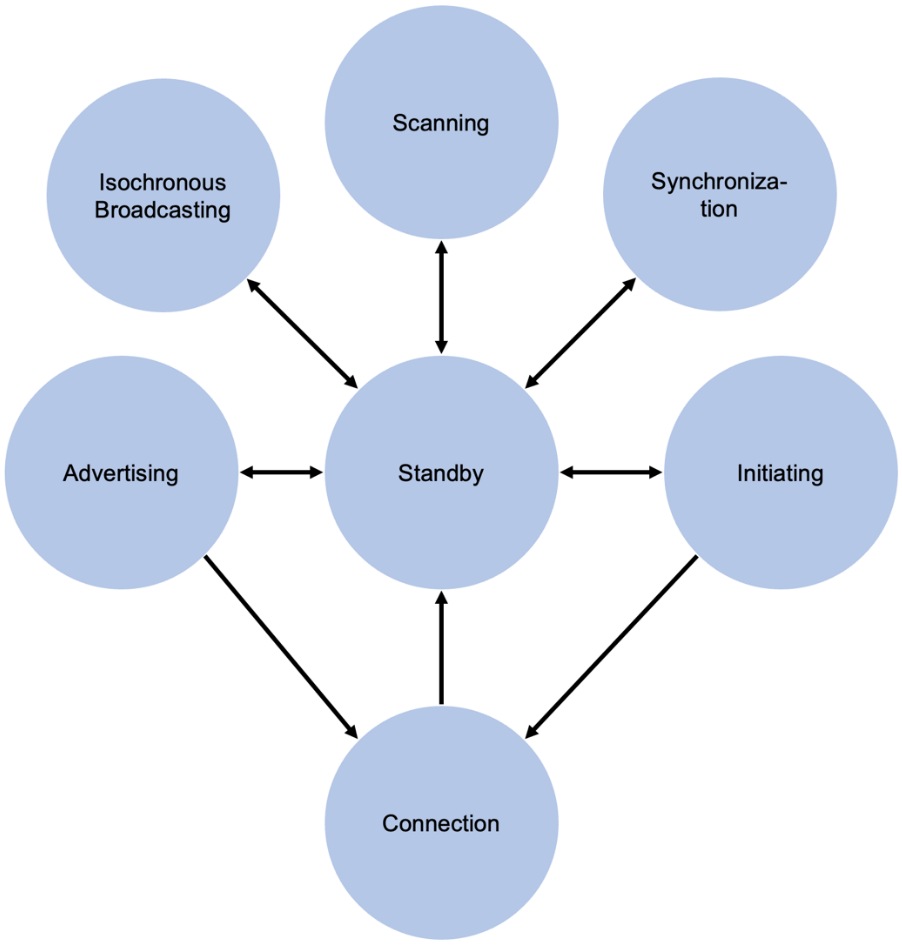

Figure 4. Diagram of the Link Layer state machine and the relation between states. (adaptive from Bluetooth SIG 2023c; Bluetooth SIG 2023b, 18)

A device’s Link Layer does not need to support all possible state combinations described in Figure 4. Nevertheless, some states require certain patterns of combinations of corresponding states. One of these combinations would be when a peripheral device uses advertising, it must have the possibility also to connect. (Bluetooth SIG 2023c.)

2.8 Host Controller Interface (HCI)

The Host Controller Interface (HCI) is a standardized layer that transports commands and events between the host and controller used by BLE and Bluetooth BR/EDR (Bluetooth SIG 2023b, 54).

In Bluetooth wireless devices, the HCI layer is built through function calls and callbacks within the wireless microcontroller. All data that communicate with the controller, such as ATT and GAP, will eventually go through the HCI layer to the controller in both directions. (Texas Instruments 2016b.)

The commands and events passing through the layer are messages exchanged between the host and the controller. Commands are sent by the host to the controller and events back to the host. An event can be a response to a command or an unsolicited message. (Bluetooth GAT 2023b, 54.)

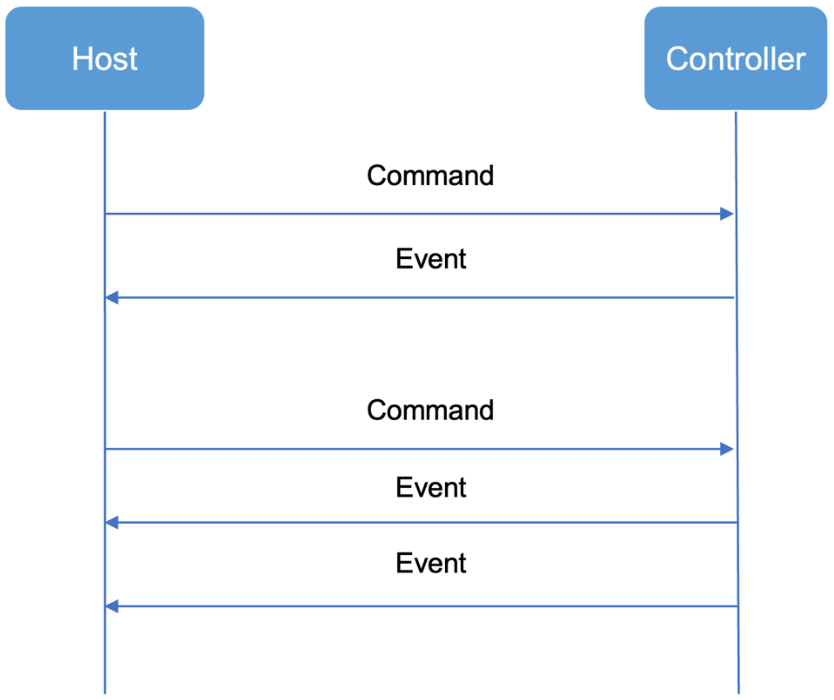

Figure 5. Visualization of the HCI layer’s commands and events (adaptive from Bluetooth SIG 2023b)

Bluetooth SIG (Bluetooth SIG 2023b, 10) describes the controller to be often something with an operating system and the controller to be a system on a chip. In common terms, the host is a mobile phone or a computer (in this project a mobile phone device) and the controller is a Bluetooth board. This project uses the Laird BT9000-SA Bluetooth board. Even if it is common to have two separate physical devices, it is not mandatory. The infrastructure works if it relies on the standard communication layer. This is crucial to allow different manufacturers to build hardware and software to work with each other’s devices.

2.9 Virtual Serial Port and UART

A serial port is a physical computer interface to send bidirectional data. This means serial ports can send and receive data. The data is transferred with an asynchronous protocol. The Laird board uses a smart BASIC Virtual Serial Port service. A virtual serial port is not a physical port as its name suggests, it is a virtual protocol that emulates a serial port. When the BL600 development board is connected to a central device, the application accepts data over the air and passes it to the UART. Any data passed to the UART also is sent to any connected central device. (Laird 2015.)

UART stands for Universal Asynchronous Receiver Transmitter, and it is a HCI transport type that may be used to implement HCI communication when both host and controller are connected on the same circuit board. (Bluetooth SIG 2023b.)

2.10 React Native

React Native is a JavaScript-based framework for building applications for mobile. As stated in its name, it utilizes native components and properties of a mobile phone, such as camera, location, and most importantly Bluetooth. Progressive Web Apps (PWA) can do the same in a browser. PWA differs from Native app development mostly by running everything in a browser. This means that even when PWA applications can use Bluetooth, a browser uses a Bluetooth Web API, which is a virtual Bluetooth utility, not a native one. It is based on React and both are built by Facebook (Meta). React Native’s advantage is the shared codebase for Android and IOS devices, which means most of the code can be used for both platforms. In comparison, Android applications run JAVA, and IOS applications are built with Swift. (Eisenman. 2017, 1.)

The clear advantage is the shared codebase with Android and IOS. The other advantages are performance (compared to progressive web apps), developer experience, and familiarity to JavaScript developers. React Native runs its UI in its own thread and not on the main thread. This way the UI feels more responsive compared to traditional JavaScript, HTML, and CSS-based web applications, such as applications with a WebView or virtual DOMs. JavaScript and even more ReactJS developers are already familiar with the React-type ecosystem. It takes less effort to learn React Native syntax and behavior compared to a JavaScript developer learning Swift, Java or Kotlin. In addition, as it is JavaScript running in a thread, React Native development tools support “hot reload”. This means that all changes are visible on a physical or virtual device right away, without bundling and building the project to a device after each edit. This cuts the development time tremendously. Also, the debugging tools are easy to use and familiar to any developer with web development experience. (Eisenman. 2017, 2.)

For React Native to be “native” to the operating system it runs on, it uses system-specific API bridges to function. On IOS, it invokes Objective-C APIs and for Android, it invokes the Java API. These accessible elements called “bridges” allow the application interface to utilize the platform’s native UI elements. In the render function, React Native’s View-component on a IOS device corresponds to an IOS UIView-element. (Eisenman. 2017, 8-9.)

3 Methods

This thesis is executed using the Constructive Research approach. Constructive Research is a process that results into an innovation and solution for a problem. It is practice-oriented and iterative which's goal is to create a new construct based on existing research. As there is a concrete output as the result, a problem, as well as a solution, is presented and justified for a real-world problem that is based on and utilizes existing theory, it is suitable to use this approach to conduct the study. (Moilanen, Ojasalo & Ritalahti 2022, 50.)

The construction of the research produced a Proof of Concept (PoC) product. During the PoC project, the company built its first-ever embedded system service. This suits the Constructive Research approach as the company does not yet have experience in the product, the product includes innovation, and unknown dependencies and obstacles. In addition, the development team does not have experience from the entire architecture. Due to these challenges, it makes sense to develop the product with lean development practices. When executed correctly, the solution unravels itself during the development and the development team plus the client will learn during the process.

3.1 Data Collection

Data collection with a lean development approach is carried throughout the development process. Information regarding the business requirements was gathered from the client stakeholders. The Product Owner typically oversees what functionalities the development team is expected to build. In this project, the stakeholder experts had a key position describing the requirements as the PO was not a laboratory employee. After all, the product is built for laboratory experts and healthcare professionals. All documentation related to the project is written and stored in the company's private document bank. The documentation includes descriptions of technical solutions, chemistry, scripts, and architecture. All relevant code and information to build the serial communication architecture is documented in this thesis in code format or plain text. The code itself acts as documentation and includes various explanations in the comments. The architecture of the system and serial communication bridges are documented as text and figures in this thesis alongside the company's private document bank.

Other data collection methods are qualitative and structured interviews with the company representatives. The requirements and feedback were gathered from the client. Conducting interviews is a data collection method to gather qualitative data and information. Case study-type projects also may require user testing alongside interviews. In user testing, it is possible to gain insight in the user's natural habitat. It helps gain unbiased information about the user when normal interviews may include hidden purposes. The interview was executed as a structured interview. A structured interview is an interview where questions are formulated before the interview by the interviewer (Moilanen, Ojasalo & Ritalahti 2022, 42-43; 81).

The client's CIO, who also acted as the Product Owner in the project was interviewed. The interview was held as a questionnaire, which was sent by email to the participant. This decision was made due to scheduling restraints. The project PO gave insight of the feedback the company stakeholders have, how the market has experienced the product, and how well the product has worked. The goal was to gain qualitative answers quickly to explain the meaningfulness of the product for the company. The interview structure is in Appendix 5 and the results are parsed in chapter "5.2 Research Reliability".

The solution/result of the research is a Proof of Concept (PoC) product. The PoC allows the company to begin running its business. The PoC is therefore used for research and development, sales, and marketing to enable future growth. The scope of the PoC was well-defined by the client in the beginning. The development team gained information about the scope and requirements by interviewing the Product Owner in the kickoff seminar. The PO also held a presentation where he explained the business and the domain for the product.

Scrum-ceremonies were held during development which ensured the team plan and gained insight into what the product needs. The Product Owner naturally was involved in giving knowledge of priorities and functionalities. Alongside the PO, there were stakeholders involved. Laboratory researchers, sales, and medical staff were always available for questions and ready to test the product after each software update.

3.2 Scrum

The project was conducted using the agile framework Scrum. Scrum was selected and requested by the client. In brief, Scrum is a framework for managing projects. It helps teams to get work done one step at a time by dividing larger tasks into small pieces. It relies on fast iteration and continuous experimentation. These result in fast feedback loops and allow the team to learn during the process. It is also a collaboration tool and one of its core goals is to bring value faster compared to frameworks such as waterfall. (Scrum.org 2023.)

The project's Scrum board acted as a flexible planner, where all the relevant tickets were created, planned, and added to the product backlog. Scrum was used in a flexible manner where tickets could be easily refined, added, and removed. It was crucial as the solution's feasibility was not understood/known in the beginning nor during the project. The flexibility allowed the project members to plan and change the plan when needed. The Scrum sprints were two weeklong events that started with sprint planning and ended in a demo for the project stakeholders. A Scrum daily meeting was held once in the middle of a sprint with the Product Owner and development team. The ceremony acted as a place to detect any obstacles and to share knowledge between development bodies. This is also where developers had the chance to interview the Product Owner about work priorities, specifications, and technical and business requirements. All information was documented to the company's private document bank and the development team created well-refined tickets to the Product Backlog describing the requirements. Scrum was not used as a textbook example, more of a lighter version of it. For example, retrospectives were ignored.

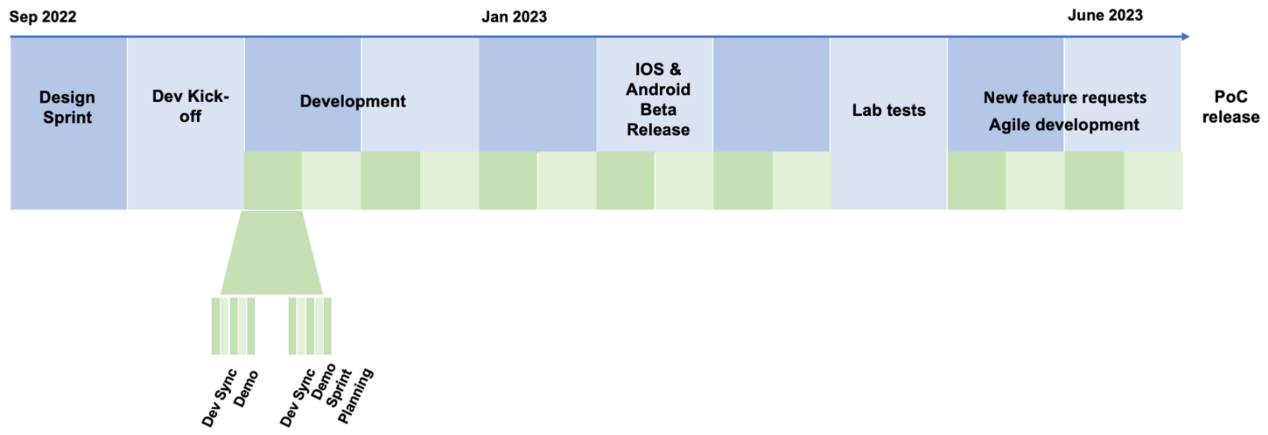

Figure 6. Illustration of the project timeline and Scrum ceremonies

Figure 6 shows the approximate timeline of the entire project, main events, and the structure of weekly event. Every Friday the development team presented their work for the entire company. This ceremony ensured all stakeholders had a possibility to witness the project's progression. After the demo, the development team and the stakeholders held a conversation about the product, requirements, and future priorities. The company also shared knowledge and news about the business and upcoming events. All relevant topics were documented in the company's private document bank and equivalent tasks were written into tickets and added to the Product Backlog.

Scrum worked well and it supported our development roadmap. Sometimes the Scrum framework itself is a chore to the developers and stakeholders. In this project, its main purpose was to hold all development ideas in one place, order the tasks by relevance and importance, and give the stakeholders insight into the roadmap. The Product Owner had visibility of has been done and what will be done next. With a small team, the sprint backlog was lightweight for us to use and plan. It was the right tool for the team to work with.

3.3 Development Process and Tests

The project followed the DevOps best practices. This means the product builds are automated and include multiple levels of automated tests. Regarding this project's scope, unit tests were written (Jest) for all utility functions. The ReactJS code currently includes 3 test suites and 43 tests. The backend has its own tests, which are out of this project thesis' scope. All tests are run automatically by the DevOps pipeline once code is committed to a branch and a branch is pushed into GIT. If the pipeline fails to build the branch due to tests, pushing the branch to the master-branch is prevented. This is to prevent harmful builds from entering quality assurance (QA) and production environments. Tests are written every time new code is introduced into the project.

After the code has been accepted in a peer code review it is pushed to the QA environment. The DevOps pipeline builds the QA environment after a commit. The developer responsible for the update checks the software is functional on all platforms. Good manners include informing the stakeholders the QA environment has been updated with the latest code. The stakeholders, more closely the scientists, did the User Acceptance Testing (UAT). This means they test the product, the key features, and make a successful measurement to ensure the product works. At this point, the code has gone through backend and frontend automated unit tests and two rounds of manual (QA) testing.

As this product is a multi-platform service, the development team and stakeholders must test it on various platforms. Each time the React Native's code is updated, the developer must make a new build of the software and release it. This means Android and IOS builds both must be done manually and released to equivalent stores for the test users to access it. Then the test users can update their applications on their mobile devices and run quality assurance testing. For both Android and IOS devices, updating the application often happens automatically or is prompted by the system. After each update, the testers needed to test by launching the application, searching and connecting to a Bluetooth device, opening the web user interface and even running the measurement flow successfully.

3.4 Key Milestones

There are a few milestones and successes worth mentioning in this project. The milestones are highly related to the communication architecture, as it was a roadblock in various ways for the product and the business. The first milestone was the possibility to search and find for nearby devices. This was a "hello world" for the whole Bluetooth capability and assured the implementation was on the correct path. Also, a "hello world" was naturally already done with the Android phone, but it was familiar technology to implement for the team. The second milestone was to successfully connect to a peripheral. Third was naturally the possibility to write to a peripheral and read the response successfully. Alongside these three core milestones, all other features were optional. The team also implemented other features, such as automatic device discovery by the device's service characteristic and automatic connection. These were more about user experience than business-critical features.

The peripheral the project used worked with an uncommon scripting language called "MethodScript". Developers were forced to study and learn the scripting language to be able to work with the device. The team studied the documentation provided by the manufacturer. Trouble arose as the scripting language is not widely spread or well known. This means additional documentation or support is widely available. The objective was to study the documentation to become familiar enough to work with the language. After all, it is a challenge to find a developer who has experience writing this rare language.

3.5 Challenges

There were multiple challenges during the implementation of the project. To begin with, selecting the correct Bluetooth version caused us to waste development time. The project began by implementing a Bluetooth Classic library first. It was selected as we noticed that no BLE devices were found when scanning for devices, but Classic devices showed up in the scanner. Another reason for this decision was the client's decision to only support Android devices in the PoC. We figured the peripheral's Bluetooth board does not support low-energy Bluetooth. After implementing a working Bluetooth Classic solution, the application managed to search, connect, read, and write data with a peripheral. Quite soon the client informed us that we must have the product for IOS as soon as possible, even if it was decided not to be supported. The team quickly realized that Apple had removed Bluetooth Classic support from their operating system and had created a subscription service called Made For iPod/iPhone/iPad (MFI). The team and the client resented such dependency and overhead. The team had to return to the plan to implement Bluetooth LE.

As the team investigated the issues finding BLE devices while scanning, they learned that the manufacturer has implemented dual-mode support for the device (both Classic and Low Energy), but the BLE features must be turned on by running a script into the physical board and enabling the functionality. After that was done, BLE devices were also discoverable.

The team faced another challenge once the connection was tried to be established. Not understanding nor having experience developing BLE infrastructures, the connections dropped in 3 seconds upon connecting. It was unclear if the peripheral rejecting or canceling the connection or if the connection was even created. After hours of research, it was determined it was a feature.

Bluetooth Low Energy in its core shuts down all connections when no data is being transferred. This is purposely to save energy. Bluetooth Classic devices maintain a continuous connection, which causes their energy consumption to be high. The team had to decide whether to propose a new connection each time data had to be sent/read or to force the devices to maintain a connection. As the peripheral had a “connected”-indicator, a physical light indicating a connection exists, and the designed UI flow was linear (connection -> measurement -> stop), we decided to do the latter. The continuous connection was built so that there is a generic listener/reader subscribed to the peripheral by the mobile device. All new subscriptions use a separate subscription with a UUID. In practice, the mobile device sends a "write"-command, starts a reader/creates a subscription, reads the response, and when done, closes the reader/subscription. Then the continuous, generic, reader in the background holds the connection and the user can inspect whether their application is still connected to the Bluetooth device before doing a new measurement. Even if it uses more energy than intended, it has not been a defect the team would notice. The device holds a charge for 1-2 weeks during development or R&D laboratory work.

4 Solution

This document has now explained the theory behind web development, dependencies, and Bluetooth and serial theory. This section of the thesis brings it all together to build a concrete solution and to successfully build a communication platform between the web, React Native, and a peripheral.

4.1 Requirements

According to the company's Chief Information Officer (CIO) and Product Owner, the company felt they had no other option than to build their own software. Neither did they want to depend on another company's product. Right from the beginning, the plan was to include a 3rd party manufacturer who supplies the potentiostat. There have been zero plans to begin manufacturing the devices themselves. Simultaneously, the company resented to be dependent on a single device or company. The company planned be flexible: it is important for them to create a service that functions with any peripheral. Also, they want to be capable to change the peripheral to another manufacturer's product. For this reason, the architecture related to the potentiostat, and the software is built as separate services. This allows either parts to be changed, modified, or even replaced.

The client had used the potentiostat manufacturer's software for R&D purposes in the past. Based on the interview with the company's CIO, they felt the manufacturer's products does not provide the same experience for their clients as their own software does. Neither could they have integrated machine learning into the product architecture, which was key for proper results. In addition, Fepod would not have control over user interfaces: their business is based on a usage-based billing strategy. This means the services are added to the user interface by service subscription. According to the PO, using proprietary software would have been impossible for their business and billing strategy.

Evaluating the client's experience and feedback of the potentiostat manufacturer's software, there where multiple flaws and lack of features. The manufacturer's application "PS Trace" is a desktop application that conducts any measurements supported by an electrochemical potentiostat. The application has a wide variety of settings and measurement types and supports wireless connections with Bluetooth Classic and USB-port connections. Howbeit, the application is built only for devices running Windows-operating system. The scripts imported to the application must be written and copy pasted by the user by hand. In addition, even if the measurement data is executed well and displayed nicely for the user, the data is not saved into a centralized cloud-based database, which was one of the key product requirements. Our client's staff copy-pasted the data by hand to their own machine and to a third-party cloud service as a file.

Alongside the desktop application there is a mobile application “PS Touch”. This application is built only for Android. Therefore, it lacks one crucial requirement as it does not support IOS-devices. IOS support is determined to be vital when entering the North American market. “PS Touch” has the same flaw considering data: it does not have a central cloud-based database. On a positive note, the application’s data format is compatible with the desktop application. Nevertheless, the data must be stored manually somewhere and imported manually to the desired application.

Neither of the applications thus has a shared database. The applications do not support IOS, Linux or Apple OS devices. The applications do not have a common cloud-based database. The tools do not have tools to save measurement-scripts to a database and Fepod can’t manage their client’s users and services.

In summary, neither application fulfills the customer’s requirements, as they lack basic modern development features, such as cross-platform usage and databases. Alongside these flaws, both applications are unstable and unpredictable, as both applications tend to crash and stop working before, during, and after measurements. The electrochemical potentiostat, built by the same company, is nevertheless ideal as it supports Bluetooth LE and custom-built sensors Fepod innovates and use.

For their business to work, a product had to be built. Based on the above requirements and the disadvantages the existing software has, it was inevitable that a tailored solution had to be built. This is when Fepod asked for help from the development team.

It was decided by Fepod that the product is built with ReactJS and React Native and it had to use Bluetooth technologies. ReactJS was selected to support both desktop and mobile users. React Native was selected to enable access to the Bluetooth module natively.

According to the PO, even when the potentiostat manufacturer’s software is not suitable for Fepod’s business requirements, the company has proven itself to be a great partner. By chance, the manufacturer’s core business model is to manufacture and provide devices for their clients: the software is only a tool that supports selling the devices. The manufacturer is more than willing to support and cooperate closely with clients who are willing to develop software and products for their devices. In early studies, the CIO and the team have already discovered future peripherals that have the potential to be used with the existing software requiring little or no development work.

The objectives of this Constructive Research project were to build a product that works on Android and IOS and can connect to and communicate with a Bluetooth peripheral device (potentiostat). The challenge was to research and develop a solution with the prescribed architecture; ReactJS must communicate with React Native and React Native must communicate with the peripheral over a Bluetooth connection. The hybrid software must work in real-time and seamlessly together.

This research does not include information about the desktop version development, nor does it include documentation about USB-serial connections or communication. This research neither explains how to build React Native nor web-applications. It is a collection of theory and practice to make an embedded system that works for any existing or new company using web technologies.

The end product is a Proof of Concept (PoC) and its objective is to allow Fepod build their company, execute research and development with their product, sales, and marketing. The solution is shipped alongside an application that works with all Android, IOS, and desktop devices.

Even if this thesis is written for a health-tech company, the solution can be adapted and scaled to any domain. In other words, it is not domain or business specific. When building new software, a company can choose to only build a React Native application which uses the Bluetooth technology to connect to a peripheral. In this case, the company should look from this thesis how communication can be done between a peripheral and React Native. More importantly, if a company has already a web application without Bluetooth enabled services, it can simply find this thesis useful to find a solution how to implement their existing product to use Bluetooth services with low effort. The final possibility is to adapt the entire solution described in this documentation and build an entire software architecture.

The product built during this thesis process became an example how to build a Bluetooth fueled application that supports all platforms. The application fulfilled the client company's requirements: supported devices include Android and IOS devices, desktop usage, user management, a centralized cloud-based database, users, roles, user access, history data and measurement scripts. It can be used wirelessly with Bluetooth LE and with a USB-cable on a browser (Web Serial API). The product enabled the client to make laboratory measurements with a handheld device. All measurement data is saved to a database and can be viewed on both mobile and desktop devices. All data can be exported from and imported to the system. The client is also able to create measurement flows and new measurement scripts directly to the database. This removes the requirement to manually insert and export scripts. From a customer's point of view, the client can tailor the experience based on their needs and do measurements without understanding scripting languages or electrochemical measurement theory.

In summary, the key reasons the company had an urge to build their own software was to have a centralized database, manage the user's access themselves, create and provide users measurements, support all crucial devices and operating systems, integrate their own artificial intelligence model, and not be dependent on a proprietary device or software.

The requirements for the application were:
- Support desktop-computers and all operating systems.
- Support Apple IOS and Android mobile devices.
- Support connectivity with Bluetooth (mobile).
- Search, connect and write to, and read from a Bluetooth Low Energy peripheral.
- Support connectivity with USB (desktop).
- Central cloud-based database with measurement history data.
- Tools to create, save, and use measurement scripts from database.
- Possibility to create and manage users (admin and clients) and tailor their service.

4.2 Technologies and Libraries

The possible variants to build a product such as this are:
- Two native mobile application (IOS and Android)
- A progressive web application
- A native application with React Native
- A native application with React Native with WebView

The first option would be to build a native application for Android and IOS. Native applications are known to be the most fluent solution, as components use the system’s own language and are optimized for the system’s hardware and software. This naturally enables the possibility to use the system’s hardware without external utilities or 3rd party libraries. The main downside is the software development time. IOS applications run Swift and Android applications run Java or Kotlin. This results into two separate projects with two separate code repositories as the code is not shareable between the two. In addition, the client desired separate tools for desktop. For a PoC, implementing all three applications takes time, effort, and money.

Mobile and desktop support could be accomplished as a progressive web app (PWA). A PWA application runs completely in the browser. The browsers support Web Serial API’s and Bluetooth API’s which can potentially fulfill the requirements. Typically, user experience suffers when running on a browser and not natively due to the fact it does not run on the operating system. Also, PWA feature support is difficult to manage as each user uses different browsers. At the time of writing, the Web Bluetooth API is an experimental project and not mature enough for production. More information about the differences between native and PWA applications listed in chapter “2.10 React Native”.

The project was decided to be written with React Native which utilizes the webview. This way code for desktop and mobile can be shared between each other. There is always the possibility to write the entire application with React Native later and remove the webview once the business is mature enough.

The project includes multiple components, and each component has their own objectives. The following list contains libraries and tools relevant to this project's success:

- React Native
- NPM: react-native-webview
- ReactJS
- Typescript
- Bluetooth Low Energy (Bluetooth >= 4.0)

In addition to the frameworks, Apple's XCode had to be used to build and ship the project for IOS devices. ESLint was used to improve developer experience, BabelJS to build bundles for development, VSCode to edit the files, GIT to manage the file versioning, Jest to write unit tests, and Azure DevOps to maintain the code repositories and product backlog. For React Native's Bluetooth communication features, the NPM library "React Native BLE PLX" was used, and it is a crucial aspect for the solution to work, although this solution is not bound to use it specifically. Any other Bluetooth library achieves the same result.

The code was separated into two git repositories: ReactJS (web) and React Native (Mobile). It was decided at the beginning of the project to treat the repositories as standalone software, as the two lack dependencies or shared code. Also, it was decided that React Native has no business logic in its code. This is not completely true, as there is currently serial validation specifically for this product. In the future, serial parsing should be done on the web completely.

4.3 Message Bridges

The end-to-end solution is a collection of message and data transfer protocols that enable sending data between components/entities. The message protocols, referred to as bridges, establish, and maintain a communication channel between two entities and enable bidirectional data transfer. These bridges must be set up between the web application, React Native, and the Bluetooth peripheral device.

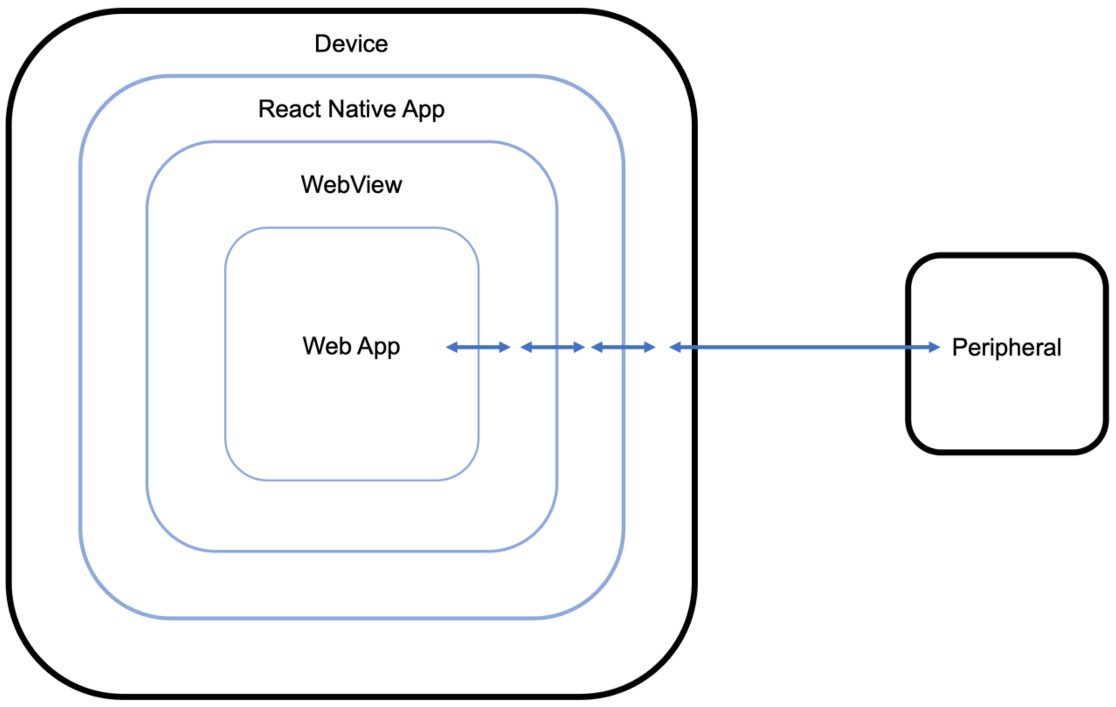

Figure 7. Required communication bridges across all entities

The challenge is to pass messages across each bridge through each system linearly in an asynchronous event. Single messages are straightforward. During electrochemical measurements, the peripheral can read and send 1 to 1000+ messages to React Native and all messages must be received, decoded, parsed, and sent to the web application. More of, the messages must be identified to relate to the correct message request. All the key features are explained in the next chapters.

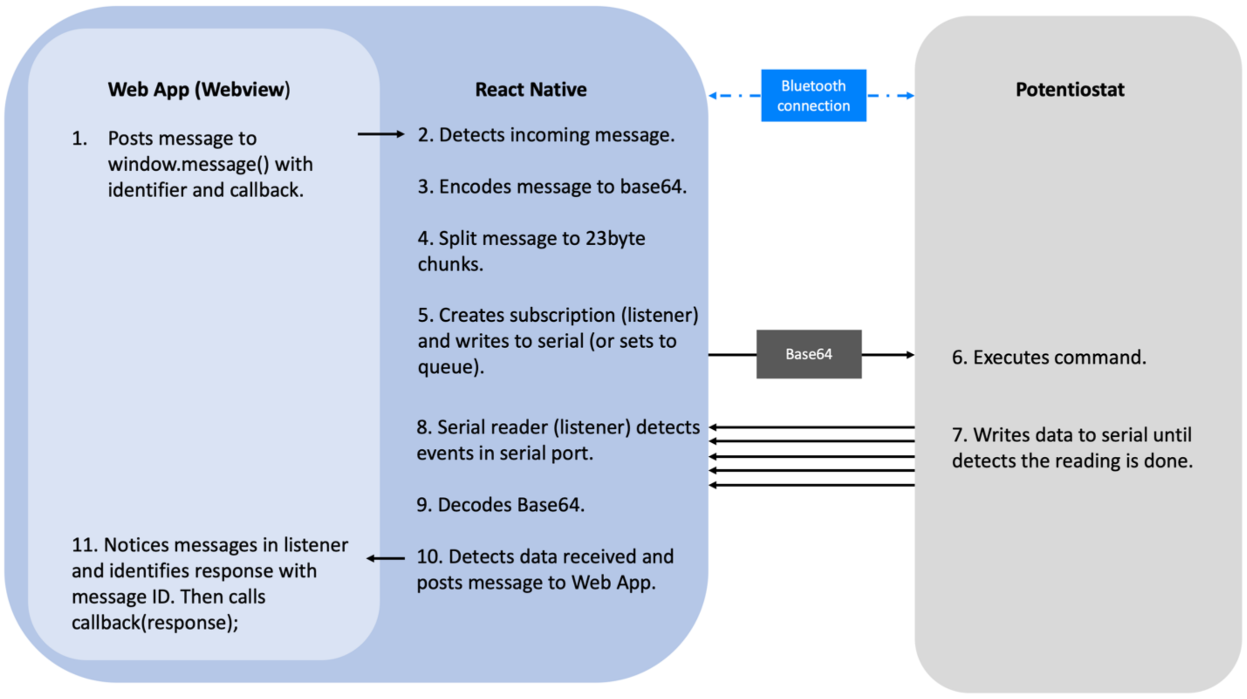
Web App (Webview)
1. Posts message to window.message() with identifier and callback.
React Native
2. Detects incoming message.
3. Encodes message to base64.
4. Split message to 23byte chunks.
5. Creates subscription (listener) and writes to serial (or sets to queue).
8. Serial reader (listener) detects events in serial port.
9. Decodes Base64.
10. Detects data received and posts message to Web App.
11. Notices messages in listener and identifies response with message ID. Then calls callback(response);
Bluetooth connection
Base64
Potentiostat
6. Executes command.
7. Writes data to serial until detects the reading is done.

Figure 8. Visualization of the end-to-end messaging across all platforms

After establishing a successful connection with a peripheral device, Bluetooth communication is ready to begin sending messages. It is suggested that the web application, ReactJS in this specific case, is the one handling all messaging requests. In other words, it is advised that React Native withholds zero business logic and acts only as a tool to manage the Bluetooth connection and passing on commands and responses. Breaking down the message thread, there are multiple steps to accomplish a successful messaging functionality. As the web application should be the one managing most of the messages, it is suitable to break down the events in a linear manner. The documentation will follow the order presented in Figure 8.

4.3.1 Sending Data from The Web-Application to React Native

The web application is hosted on a separate server, so it is obvious it has the same features any web application has. Therefore, the web application has a window-object. This object has a property "window.ReactNativeWebview" which has the same "postMessage"-function. It can be used to send messages to the "webView"-reference. The NPM package "react-native-webview" is responsible for adding the object property to the window-object. The property "window.ReactNativeWebview" has the property "onMessage" property which is used to send messages to React Native.

```
export const postMessageToRN = ({
  type,
  payload,
  callback,
  id = uuidv4(),
  endOfResponseCharacter,
}: MessagePayload): void => {
  window.message({
    type,
    payload,
    callback,
    id,
    endOfResponseCharacter,
  });
};
```

If writing Typescript, it is obvious the code bundler does not know “ReactNativeWebview” property exists in the window object as it is added by the system. A type-definition file should be created, and respective types added:

(global.d.ts)

```
export {};

declare global {
  interface Window {
    message(message: MessagePayload): void;
    ReactNativeWebView: {
      postMessage(message: string): void;
    };
  }
}
```

To receive data that is relevant to the message sent, it is important to add a listener and listen to the messages. As React Native is used for both IOS and Android devices, it is important to support both. IOS devices use the window-object, hence Android devices use the document-object when listening to messages. The callback function called after receiving message is specified and passed as a function parameter from the client side. The web application must specify “window.message” as a function. As seen in Appendix 1, the function is declared in a project root-file, preferably in the same file as the application router or index.js-file. The function variables are specified by the programmer, but in this example, the payload is throughout the application as follows:

```
export enum MessageTypes {
  WRITE = 'WRITE',
  READ = 'READ',
  ALERT = 'ALERT',
  SET_DEVICE_NAME = 'SET_DEVICE_NAME',
  SET_SHOW_WEBVIEW = 'SET_SHOW_WEBVIEW',
  SET_LANGUAGE = 'SET_LANGUAGE',
  SET_IS_DEMO_MODE = 'SET_IS_DEMO_MODE',
}

export type MessagePayload = {
  type:
    | MessageTypes.WRITE
    | MessageTypes.READ
    | MessageTypes.SET_DEVICE_NAME
    | MessageTypes.SET_SHOW_WEBVIEW
    | MessageTypes.SET_LANGUAGE;
  payload: object | payload | string; // i.e "t".
  callback?: (data: unknown) => unknown;
  id?: number | string;
  endOfResponseCharacter?: EndOfResponseCharacter;
};
```

“MessageTypes” are application specific and pre-determined message types. Required types ought to be “write” and “read”. “Payload” is the message body. “Callback” is a callback-function to be called after each message received (appendix 1). “ID” is used to specify a custom identification. If no identification is provided, the message-function should create a UUID.

The “endOfResponseCharacter”-property is business-specific and relates to the peripheral. It is used for the client application to know when a command request is complete. If the client can expect only one response, it is not needed. Once the response is received or the client determines that the reading process is complete, it closes the reader/listener. Closing all subscriptions and listeners is crucial. If left active, responses are detected arbitrarily.

It is advised to create a utility function that the client can import and call anywhere in the project. This way the variable types can be specified and set without calling the global window object directly, even if it is possible to call it directly.

```
/**
 * General utility to post message to React Native.
 */
export const postMessageToRN = ({
  type,
  payload,
  callback,
  id = uuidv4(),
  endOfResponseCharacter,
}: MessagePayload): void => {
  window.message({
   type,
   payload,
   callback,
   id,
   endOfResponseCharacter,
  });
};
```

Breaking down the function of sending messages further (Appendix 1), there are a few things to understand. Once the message is sent to “window.ReactNativeWebView.postMessage”-function, there must be a listener created to receive messages back. IOS and Android devices use different properties to send and read messages. IOS uses the “window”-object and Android uses the “document”-object. “globalObjectToUse”-variable chooses the correct property to call when reading values. The pattern of the listener’s callback is important: the second variable for the listener function is the callback function. It must be specified as a named function. Otherwise, it is out of scope and the listener cannot be destroyed with “removeEventListener”-function. The rest of the function is straightforward as the listener detects messages and evaluates the message ID being the same as this listener’s ID. If the response is considered complete, the listener is removed.

4.3.2 React Native Receiving Messages from The Web Application

React Native has similar functions as the web application. The WebView is declared and is a key property creating the nodes for successful communication. The three crucial properties for the communication to work between the WebView and web application are the component reference (ref), source, and “onMessage”-event property. (Appendix 2.)

“Source”-property is the web service’s URL. “webViewRef” is the “reference”-object to the “WebView”-component and is used to send messages back to the WebView. More about it is explained in chapter “4.3.3 Sending Data from React Native to the Web-Application”. Important when receiving messages coming from the web application is the “onMessage”-property. It requires a function, and that function is used as a callback for any incoming message detected by the “WebView”-component. In appendix 2 there is a callback function “onMessageService” and it has the commonly used event-variable. “event.nativeEvent.data”-object contains all the information the web application has posted to the window-object. As the data is always the type “string”, it must be parsed to be a JavaScript object. As described in the previous chapter, data is expected to be an object with the properties type, payload, id, and endOfResponseCharacter. Once the message’s data is parsed, it is up to the developer to decide what to do next. It is advised to call a dispatch function with pre-determined actions. These actions are related to the previous chapter’s action-types. Example of a dispatch function in ReactJS:

```
/**
 * Received messages middleware.
 */
const onMessageDispatch = useCallback(
  async ({ type, payload, id, endOfResponseCharacter }: IMessage) => {
    switch (type) {
      case MessageTypes.SEND_MESSAGE:
        postMessage({ type, payload });
        break;
      case MessageTypes.WRITE: {
        // Write(command) must be a string.
        await actions.write({
          command:
            typeof payload !== 'string' ? JSON.stringify(payload) : payload,
          id,
          endOfResponseCharacter,
        });
        break;
      }
      case MessageTypes.GET_CAMERA_PERMISSION: {
        requestCameraPermission((isGranted: boolean) => {
          postMessage({ type, payload: { data: isGranted, id: id } });
        });
        break;
      }
      case MessageTypes.OPEN_APP_SETTINGS: {
        openSettings();
        break;
      }
      default:
        break;
    }
  },
  [actions, postMessage, i18n],
);
```

To read and write information to a Bluetooth peripheral, at least dispatch types should include actions “READ” and “WRITE”. The functions the actions should call depend on what Bluetooth library is used and how the developer has written it. In this example, the entire Bluetooth manager is written as a “React Context” (Appendix 3) and can be used as a guideline.

4.3.3 Sending Data from React Native to the Web-Application

Once the data is received from the peripheral or React Native should send system messages to the web application, React Native requires a way to send data to the web application. It is done through the “WebView”-component properties. The “WebView”-component must have a reference for incoming and outgoing messages for it to work. The “useRef”-hook creates a reference object that declares a ref (appendix 2). It is useful to create a reference to a value with a reference when the value is not needed to be rendered in the DOM. The ref-object has a property ”ref.current” that is mutable, but it will not re-render the virtual DOM (Meta Open Source 2023).

The reference’s property ”current” includes a key-value pair: “postMessage”. It is a function that allows posting messages to the web applications window-object (Appendix 2). Mozilla Developers (MDN 2023a) explains that “window.postMessage()” is a method that enables safe cross-origin communication between window objects. This is important as typically cross-origin scripting, meaning two websites with different hosts, are disallowed from sending messages to each other by the browser. The reason behind this is security. The message sent must always be a string type.

4.3.4 Receive Messages in Web-Application Coming from React Native

As stated in chapter “4.3.1 Sending Data from The Web-Application to React Native” the post-message function expects messages and handles the response messages by itself with a request ID and callback. In the case of unsolicited messages, messages that the web application receives without sending a message request first, there should be a global listener on the top level in the application.

In case of unsolicited messages coming from React Native to the web application, it is useful to have a common listener on document load attached to the top level of the document.

```
// index.tsx (web app)

useEffect(() => {
  /**
   * Listen for messages coming from React Native.
   * This should be used only for messages posted from React Native,
   * which ReactJS is not expecting. Otherwise use a callback.
   *
   * Note: "document" for Android, "window" for IOS.
   */
  if (navigator.userAgent.includes('Android')) {
    document.addEventListener('message', (event: MessageEvent) =>
      onMessageReceived(event)
    );
  }

  if (navigator.userAgent.includes('iPhone')) {
    window.addEventListener('message', (event: MessageEvent) =>
      onMessageReceived(event)
    );
  }
}, []);
```

IOS and Android-specific variables apply also in this case. IOS uses the Window-object and Android uses the Document-object. It is beneficial to also check the domain messages are coming from by specifying the target source. This is useful to prevent any harmful domains from sending cross-origin messages.

4.3.5 Base64

Base64 is a binary-to-text encoding that represents binary data in an ASCII string format. The text is translated into a radix-64 representation. Base64 encoding is commonly referred as encoding binary data for storage or in media data transfer that can only deal with ASCII text. This helps encode and decode the text reliably without modifying the source text. (MDN 2023b.)

It is important to remember that the peripheral device specifies if its communication is done in UTF-8 or encoded text, such as base64. The potentiostat uses base64 encoding to read and write data over Bluetooth LE. In addition, the Maximum Transmission Unit (MTU) for the peripheral is set to 23 bytes. This means that the device can only receive a maximum of 23 bytes at a time. In other words, all messages sent for the peripheral must be less than or equal to 23 bytes and in base64 format for this project.

The messages must be split into chunks of the size of the device's MTU (Appendix 4). Then the messages are sent to the device one row at a time until all commands are sent for the device using the "write"-characteristic. On the contrary, all the data received by the listener in React Native should be decoded into human-readable UTF-8 format and joined together to create a complete command. The completeness of a row depends on the expected data and is case specific based on the peripheral.

4.4 Bluetooth Utilities

There is numerous 3rd party NPM libraries to manage Bluetooth communication available. It is also possible to write one for any project yourself. This project was written using a commonly used

“React Native BLE PLX”-library. The library has over 12,000 downloads weekly, at the time of writing. It is easy to use and well-documented once the core functionality of the Bluetooth LE is mastered.

This section goes through the core principles of how to create a Bluetooth connection between a mobile device and a peripheral and how to read and write with a peripheral. (Appendix 3.)

4.4.1 Bluetooth Permissions

Connecting to a Bluetooth device requires permission from the user. There are a few differences between IOS and Android devices.

For Android, the permissions required by the software must be declared in the Android’s manifest (AndroidManifest.xml). It is a file found in Android projects and can look like the following:

```
// android/app/src/main/AndroidManifest.xml

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.fepodmobileapp">

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.BLUETOOTH"/>
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
    <uses-feature android:name="android.hardware.bluetooth_le" android:required="true"/>
    <uses-permission android:name="android.permission.BLUETOOTH_SCAN"/>
    <uses-permission android:name="android.permission.BLUETOOTH_CONNECT"/>
    <uses-permission-sdk-23 android:name="android.permission.ACCESS_FINE_LOCATION"/>

    <application
      android:name=".MainApplication"
      android:label="@string/app_name"
      android:icon="@mipmap/ic_launcher"
      android:allowBackup="false"
      android:theme="@style/AppTheme">
      <activity
        android:name=".MainActivity"
        android:label="@string/app_name"
        android:configChanges="keyboard|keyboardHidden|orientation|screenLayout|screenSize|small-
estScreenSize|uiMode"
        android:launchMode="singleTask"
        android:windowSoftInputMode="adjustResize"
        android:exported="true">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
      </activity>
    </application>
```

```
</manifest>
```

It is important to remember that Bluetooth LE requires permission to location (ACCESS_FINE_LOCATION or ACCESS_COARSE_LOCATION) alongside the Bluetooth permissions. This is due to Bluetooth LE data containing information that could be used to track the device's location.

Then each time the application is opened the permissions are asked or checked by a separate function. This means there needs to be a function in the code that prompts the user permissions at least once, i.e.:

```
/**
 * Request phone for application to use location and bluetooth services.
 */
export const requestBluetoothPermissions = async (
  callback: PermissionsCallback,
): Promise<void> => {
  try {
    if (Platform.OS === 'android') {
      const result = await PermissionsAndroid.requestMultiple([
        PermissionsAndroid.PERMISSIONS.BLUETOOTH_SCAN,
        PermissionsAndroid.PERMISSIONS.BLUETOOTH_CONNECT,
        PermissionsAndroid.PERMISSIONS.ACCESS_FINE_LOCATION,
      ]);

      const isGranted =
        result['android.permission.BLUETOOTH_CONNECT'] ===
          PermissionsAndroid.RESULTS.GRANTED &&
        result['android.permission.BLUETOOTH_SCAN'] ===
          PermissionsAndroid.RESULTS.GRANTED &&
        result['android.permission.ACCESS_FINE_LOCATION'] ===
          PermissionsAndroid.RESULTS.GRANTED;
      callback(isGranted);
    } else {
      // For IOS.
      callback(true);
    }
  } catch (e) {
    console.log('Error requestingPermissions', e);
  }
};
```

For IOS, there is no need to ask permission with a separate function. Instead, IOS expects all application permissions to be added to an "Info.plist"-file similar to Android's Manifest file:

```
// ios/FepodMobileApp/Info.plist
...
<key>NSLocationWhenInUseUsageDescription</key>
<string>Location is required to search for nearby bluetooth-devices.</string>
```

```
<key>NSBluetoothAlwaysUsageDescription</key>
<string>Bluetooth is required to connect a potentiostat.</string>
<key>NSBluetoothPeripheralUsageDescription</key>
<string>Bluetooth is required to connect a potentiostat.</string>
...
```

After the user allows the device to use Bluetooth functionalities, it is possible to scan devices with a scan function using the Bluetooth manager. It is possible to use the device's service UUID to scan specific devices or all nearby devices. (Appendix 3.)

As explained by Texas Instruments (Texas Instruments 2016a), The GAP layer initiates the device discovery, scanning, and connection procedures at this point.

4.4.2 Writing to A Bluetooth Device

As stated by Afanef and Bluetooth SIG (Afanef June 2018, min. 4-5; Bluetooth SIG 2023b, 61) the Generic Attribute Profile (GATT) is responsible for services and characteristics, which enable the possibility to write to a peripheral. At this point, we know that characteristics are 128-bit or 16-bit UUID values (Bluetooth SIG 2023b, 68). The characteristics must be either discovered with the Bluetooth manager or known based on the peripheral's documentation.

The NPM package BLE PLX includes a "discoverAllServicesAndCharacteristics"-function that can be used to discover available characteristics. It is advised to discover characteristics and services once upon connecting to a device to exchange information and protocols. Once a characteristic, such as the "read"-characteristic, is known, it can be declared as a constant: it never changes (unless a software update changes it manually). Example can be found in Appendix 3.

The write command is simple:

```
await device.writeCharacteristicWithResponseForService(
  SERVICE_UUID,
  WRITE_CHARACTERISTIC_UUID,
  command,
);
```

There are three required variables: the device's service UUID, the write-characteristic, and the text to write (command). The device service UUID can be asked from the peripheral alongside the "discoverAllServicesAndCharacteristics"-function, or it can be known based on the device's documentation. That can also be written as a constant as it does not change.

It is best to understand that a serial port can only handle one writing process at a time. At least it is best to consider it that way even if it is possible to push multiple commands simultaneously. The issues arise as soon as the reader does not know what its responses are and to which callback the responses belong to. The best way to handle multiple simultaneous “read”-events is to keep a state of the active reader. In this example, the subscription is saved as a variable with the React hook “useState()”. The subscription includes a function to cancel a subscription, so it is beneficial to save the subscription object to the state. As one subscription is active at a time, all occurring “write”-commands are then saved into a queue. Each time a subscription is canceled, the next item in the queue should be called until no more subscriptions exist in the queue. (Appendix 3.)

4.4.3 Reading from A Bluetooth Device

Now that the writing process is covered, the reading process follows. As stated, there should be a subscription, serial event listener, created during writing. Like writing to the write-characteristic, reading the messages requires monitoring a characteristic. This time it is the “read”-characteristic.

```
const subscription = device.monitorCharacteristicForService(
  SERVICE_UUID,
  READ_CHARACTERISTIC_UUID,
  (error, value) => {
    // Do something.
  }
);
```

There are three required variables for a successful read: service UUID, the “read”-characteristic UUID, and a callback function. In this example, there is declared an unnamed function. The unnamed function’s second variable “value” holds the characteristic value. This value is the value the device sends for the subscription to read and contains the response data. (Appendix 3.)

For React to understand something has changed or data from the device has been received, it is advised to add the read-data to a state. As the state changes, it is possible to let React Native know there is data to be passed on to the web application. In this example, as the entire Bluetooth communication module is built with React Context, there is a reducer dispatch-event that updates the “payload”-value of the reducer state. It is up to the developer to decide should the entire serial request first be gathered and sent as one message forward or send data row by row as soon as the serial port has provided a value. In this project, each row is sent one by one. The “_writeCallback”-function updates the “read”-state when reading a characteristic value and the React Native application sends the message to the WebView with the “postMessage”-function. (Appendix 2; Appendix 3.)

Now we have built all the bridges required to build a successful loop of events. The web application will then detect the incoming messages coming from React Native and as the request provided a UUID which was added to the “write”-subscription and passed to the “onRead”-state and back to the web application alongside “postMessage”-function, the web application can match the request with the response. As stated before, it is advised to have one read subscription active at a time. Nevertheless, with this structure, the web application can create multiple requests and message listeners. The UUID and the asynchronous nature make it possible, and a developer can always rely on messages being sent in chronological order.

5 Conclusions

Now this document has displayed the necessary theory about the Bluetooth technology, web applications, and React Native development. The document also explains how to build communication bridges across all three entities. This chapter will recap the research questions and summarize the key findings related questions.

5.1 Research Questions

The initial goal and key indicator of the development project's success was tied to the research questions. The main objective was related to the first research question (RQ1): "What are the requirements for enabling Fepod to start their business?". To allow the company to begin its research and development work in laboratory studies, the product had to be released and shared with the company. Also, the product had to be stable and work standalone without developer assistance. Alongside R&D activities, the company had to be able to demo the product to buyers and investors.

The mobile application was released quite soon at the beginning of the development process. The R&D team gained access to the product during the first 3 months of development. This allowed the R&D team to use, investigate, gain experience, and support the development team. After 4 months of development, they could integrate their pre-existing data onto the system and begin working on data analytics and do measurements in their laboratory. Already right after the alpha version, investor candidates were invited to meet the team and it was possible to present them working versions of the service and product. When the PoC was completed, the company attended multiple seminars and conferences where successfully demoed the product without assistance from any developer.

In this regard, the project was victorious. The research question was answered as the company did not have a product to demo, but they had an idea of the R&D and a business model. The PoC product allowed them to start the business. The product has been online and stable since June 2023 and has required closer to no actions from developers to function. This includes both web and mobile applications. In addition, the R&D results have been positive: the test samples that were used to evaluate the measurement results returned expected values in a controlled environment. The company was able to make laboratory tests and point-of-care measurements with the system and existing equipment, such as slips. The company is now working on marketing and sales as planned and expected.

The second research question was a blocking issue for technical development (RQ2). It was known the company requested the development team to build the software with React Native. However, it was undetermined if communication between React Native, the peripheral, and the web would work. The team had zero experience, nor where there evidence the architecture works to fulfill the business's needs. Simultaneously, this functionality is at the core of the project's success.

There is evidence and technical documentation about the different components separately. This project was a research and development effort, which combined all components together to work as a seamless architecture. In the beginning, there was no documentation found to prove it works. The project is meaningful as it proved it is possible, feasible, and plausible. This solution also explains the dependencies required to bridge all nodes together seamlessly.

The solution worked to support the business needs. The communication between all platforms happens in real time with message events and virtual serial ports. The measurements and queries required by the business-related peripheral support the business requirements. The “window”-object's messages-event works in favor of real-time and chronological communication. As does the serial. As all communication is done through the same events, the identification (UUID) and subscription-based listeners have proved to embrace the purpose in an infrastructure where all the entities send and receive solicited and unsolicited messages. It was important to the client to have a product that runs on all three platforms: IOS, Android, and desktop browsers. The PoC covered all three interfaces and devices. In addition, the success was shipping the product for demo and test users via application stores early on.

Considering the third research question, how other companies can adapt this solution to their own business (RQ3): according to Bluetooth SIG's 2023 Bluetooth Market Update (Bluetooth SIG 2023f), the growth in Bluetooth devices will continue to grow linearly. The report predicts a 9% annual growth in the number of Bluetooth devices during the next 5 years (until 2027). This means that today 5.4 billion Bluetooth-enabled devices are shipped yearly, 7.6 billion devices are shipped in the year 2027. There is a large existing market for Bluetooth communication, products, and devices.

Bluetooth is also advancing. New features are built by the Bluetooth SIG community all the time. This indicates Bluetooth is not becoming obsolete anytime soon. Audio devices have used Bluetooth Classic, but Low Energy audio has become available now. This opens possibilities to use Bluetooth communication protocols with other health technology, such as hearing aids. Bluetooth SIG report also explains sports trackers, wellness and sleep devices, computer accessories, blood pressure monitors and others will grow. IoT devices also continue to grow due to mass manufacturing costs. Also broadcasting devices, such as indoor navigation, tracking, item finders, digital home and car keys, monitoring systems and warehouse shelf/item labeling are becoming more common. (Bluetooth SIG 2023f.)

All the aforementioned markets are growing based on the study. Each market and Bluetooth product requires a connection to a central device. The architecture used in this thesis provides a reliable, cheap, and easy-to-implement solution for quick results. It does not only cover new products. All existing web applications can be upgraded to use Bluetooth features. Any existing or new web application can be added to a React Native WebView with the Bluetooth controller (Appendix 4). In addition to Bluetooth features, the product is ready to be shipped for IOS and Android devices at the same cost. Any existing product that uses Universal Serial Bus (USB) and has requirements to be an embedded system can adapt to be wireless and used on a mobile device. The solution is not domain-specific and there are no dependencies in any business domain. Still, exploring only health-tech, as the project was a health technology product, Healthtech Finland (Teknologiateollisuus 2023) explains the health technology product exports exceeded 2.7 billion euros in the year 2022. Exports grew by over 6% over the previous year.

5.2 Research Reliability

During the Constructive Research and project development, validation was conducted as described in 4.3 Development Process and Tests. The unit tests and DevOps practices demonstrate and ensure the project's maturity, but more importantly the user acceptance tests (UAT) executed by the stakeholders ensure the projects validity. The UAT is the highest remark in a project as it implies the feature is completed up to the standards expected by the requester. It also means the briefing, planning, implementation, and release of the feature was successful. A successful release of a feature includes naturally a successful build through the pipeline and tests and all these aspects ensure the product lives up to the standard set by the stakeholders. All these processes also ensured the project fulfils the business requirements.

After the project ended, an interview was conducted to gain insight into the impact, relevance, and importance the project had for the company. The interview was executed as a quantitative data-gathering structure. It was a structured interview; the questions and the order of the questions were pre-determined by the interviewer. Due to scheduling constraints, the interview questions and answers were handled via email. The interviewee is the company's CIO who also acted as the project's PO. The structured and qualitative interview format was selected due to the fact there was not a high number of stakeholders that have vast experience with the project. The objective was to gain comprehensive insight into the project execution and the results/impact the project had on the company.

Regarding their bias, interviewee may be inclined to treat the project as a success. He is responsible to answer to the stakeholders, the CEO, and the investors about the project and the project's budget. It may be negotiable, but one must trust the interviewee is legitimate with their answers. The limitation of the interview is the fact the interview was not held in person. The feelings and environment of the interviewee are absent in these results and cannot be described.

In the future or similar projects, it would be beneficial to interview a variety of stakeholders. Without time constraints, the interviews could be even held as group interviews. The feeling and inspection of biases could be inspected during the interview. It could provide more evidence of the mindset and objectives interviewees have.

The interview questions are found in Appendix 5 "Interview Questions". The questions were predetermined and sent in the same order described in the appendix. The following sections are the parsed answers of the interview.

Regarding whether the PoC product provided a solution to a problem the company faced, the interviewee stated "Absolutely". They mention the PoC being a reliable and stable solution. As they had the challenge of being able to communicate with the manufacturer's peripheral device and a mobile phone, this product made it possible. The interviewee also mentioned there were other options on the table in the beginning, but this solution ended up being the best one to support their requirements.

Considering key objectives, the interviewee explains that reliability was key. The communication architecture between the peripheral, mobile device, and web application acted as a key feature and the solution fulfilled its purpose as planned providing a reliable and stable user experience and data flow. Another key objective mentioned was to have a solution that works with numerous devices, manufacturers, and customer and partner companies. The interviewee mentions that the solution is capable of being adapted to any product, peripheral, or company with no or little modification. The interviewee feels that the key objectives were met as the communication architecture and supporting multiple devices were successfully built/enabled.

The interviewee had no evidence and was resentful to say the product has a meaningful impact on the health-tech industry, as it is extravagant to say something like that. Especially with so little time for product testing and sales. What the company can say is that the peripheral manufacturer is impressed with the software Fepod built. They mentioned the architecture and the communication functionalities across the product are impressive. The company even expressed that they are interested in adapting this software for their other clients.

The interviewee describes the milestones for the project. For the project to be successful, numerous milestones had to be met and challenges resolved. The most crucial one was building the Bluetooth connection and enabling communication between the Bluetooth peripheral and the mobile device. In the early stages of the project, searching for devices and connecting to devices was crucial to work. The next milestone was to write to and read the serial ports.

Regarding the project's development processes, the interviewee mentions the project used Agile development methods. A backlog existed where the team could add development ideas and requirements. Ideas were categorized into different levels of entities custom to Scrum, such as epics, features, user stories, and tasks. The tasks were used to build two-week sprints and the work was carried out by the development team from the sprint backlog following Scrum's best practices.

The interviewee mentions the feedback from the users has been highly positive. Establishing a connection with the potentiostat device and mobile phone is easy to use and the data transfer from the peripheral to the database is reliable. The company has tested the communication architecture and application for hundreds of hours and has made thousands of blood measurements. Issues, bugs, and errors are rare. The team has expressed the PoC product has fulfilled their needs, and no immediate development work is required.

When asked about the positive impact the product has had on the company's business and how has it been visible, the interviewee mentioned the features of the product are at the core of the company's business model. This means the product enables the company to begin its business and sales. If the key features in this product had failed, the company would have been forced to rethink numerous pivotal aspects, such as which potentiostat, central devices, or data transfer techniques/architecture to use. Now that the PoC is done, the company may begin focusing on sales and marketing. The interviewee explains the product is to become a commercial product set for sale. The PoC has already proven to create value in the company for R&D purposes and has pushed development efforts in the peripheral manufacturing company.

When asked if this product had not been done if the project would have not been successful, and what other options the company would have had, the interviewee explained they did not have other options on the table or even desire to use any other software than their own. It was planned from the beginning to use a 3rd party peripheral. Fepod never considered manufacturing a device themselves. The PO mentions they do not want to be bound to any manufacturer, but to remain flexible and keep the possibility to support multiple devices and manufacturers. He mentions that therefore the bridge between the device and the software is built as separate constructs. Parts of the service can be changed without altering the entire product.

The company had the possibility to use the peripheral manufacturers software. In that case, Fepod would have not been able to

- provide the same level of user experience for the end users,
- integrate machine learning,
- or manage themselves how our products and services are used as a usage-based billing model.

Nevertheless, PalmSens (the peripheral manufacturer) has proved themselves as a great partner as their core business happens to be manufacturing and selling the devices. The PO explains their software is not their core business: it exists only to support the sales of the devices. PalmSens supports actively partner companies who are willing to build their own software and solutions for their devices. In the future, Fepod is looking into other peripheral devices from all companies. Devices with different purposes, new measurements, and lower costs.

The interviewee explains the project was overall a success. Even when the project required knowledge of multiple topics and skills from several technologies, the development effort was straightforward, and it was a joy to work with the development team.

6 Discussion

Previous chapters, especially chapters 3, 4, and 5, cover a lot of the project's outcomes, challenges, key findings, and lessons learned. The following chapters portray the writer's thoughts about the project and provides insight and perspective of the businesses and project's future.

6.1 Project Reflection

During the development process and throughout the project I learned project management, development, and technical research skills. Working with a small and agile project differs from a project with medium or large multi-skilled teams. This meant team members had multiple responsibilities. The frontend for this project meant developing a ReactJS web application, an Android and IOS application with React Native utilizing Bluetooth LE functionalities. Web development itself withholds various techniques, such as APIs and component library management. In addition, Typescript experience was required. Including the project management skills, it takes an experienced developer to keep the strings tied together.

The scope of the project was rather fixed with room to scale due to the well-defined scope for a proof-of-concept product. It was not clear the architecture would work just as planned. It was important to have room for upcoming and most importantly unpredictable events, such as sudden requirements for IOS support and challenges with Bluetooth Classic and LE. The project's flexibility allowed the project to succeed in that regard.

I was skeptical about how well the React Native's "Webview"-component will perform. Around 2016 I used Webview in a project and the performance was not ideal: it was slow, sluggish, and interaction felt like a poor user interface. Clicking the UI elements worked only after a delay. At the beginning of the project, we had a conversation about this concern with the project architect. It was discussed should the entire project be separately done for mobile and desktop. I did not see that as a big investment as the team is experienced. We decided to try it how it was planned first, with the Webview. The "Webview"-component has improved drastically during the last years. The performance has increased greatly. In my opinion, an average user no longer notices the interface not being a native application. It works seamlessly between React Native, and the web application and the user interaction is smooth and fast with closer to no lag. I would recommend using the React Native Webview in the future and would use it again in a future project if necessary.

Once the architecture was planned, many developers questioned the hybrid solution and asked why the application was not built separately for mobile and web. I had the same thought. For experienced developers building two platforms is no chore. I figured, as it is not a chore, nothing is preventing the company from writing the applications separately someday now that the PoC is done. The admin tools would then be on the web and measurement tools on mobile and the web separately. I believe the approach of building it with a WebView was justified: some user interfaces are shared with mobile and desktop-sized displays. There is no idea writing duplicate work before the business has proved itself in the market. The user interface has also been built with a JSON renderer. The JSON markup is fetched from an API and rendered with equivalent elements. This ensures that even if the renderers are platform-specific code, both mobile and desktop applications can use the same source of JSON someday. As mentioned before, there is no hurry to build the separate applications as the WebView and web application works so well on mobile.

6.2 Suggestions For Future Development

As mentioned, the PoC is now under research and development work in laboratory experiments and the company implements sales and marketing strategies. During the project, alongside implementing the Bluetooth communication tools, the team also implemented USB serial communication features. This means the same peripheral device can be connected to a computer with a USB-cable. The exact same application and same code can be used to make measurements directly with a computer with a wired connection. This is implemented using the Web Serial API: it is a web tool supported by limited internet browsers and it enables the possibility to read serial data from the machine's native USB-port. Even if the client application shares the same code, the Web Serial API works in a slightly different way. It requires some familiar, but still unique code to work alongside the Bluetooth implementation.

One feature worth researching and implementing is the Web Bluetooth API. It is like the Web Serial API, but it enables the web browser to interact with Bluetooth LE devices. It is in the experimental stages of development, but it could enable some possibilities for working with desktop computers and Progressive Web Apps (PWA). The future will tell if it will be standardized for web development or if it would be better than native solutions.

In addition, the company is looking into supporting a wide variety of peripherals. The idea of independent products and genericity is in the core of the planned PoC and in the outcome of the project as it was also explained in the project requirements. With the same software and serial communication architecture, it is possible to change the peripheral device or add measurement types endlessly. The Bluetooth protocol standards ensure any device supporting Bluetooth LE can be connected to the software. In theory, any scripts and script formats can be sent to the device with only new configuration and command syntax. This includes IoT, health-tech devices, smart home, assistant living for the elderly, medical equipment, any biometric sensors, glucose monitors, activity, and motion sensors. Based on the interview with the project’s PO, the company is already testing and reviewing other peripheral devices and use cases. Based on the feedback, the product works already almost from the box with additional devices.

References

Adams, S., Doeven, E., Quayle, k. & Kouzani, A. 2019. MiniStat: Development and Evaluation of a Mini-Potentiostat for Electrochemical Measurements. IEEE Access. Vol 7.

Afanef, M. April 2018. Ellisys Bluetooth Video 1: Intro to Bluetooth Low Energy. Ellisys video. URL: https://www.youtube.com/watch?v=eZGixQzBo7Y&t=464s&ab_channel=Ellisys. Accessed: 13 October 2023.

Afanef, M. June 2018. Ellisys Bluetooth Video 5: Generic Attribute Profile (GATT). Ellisys video. URL: https://www.youtube.com/watch?v=eHqtiCMe4NA&ab_channel=Ellisys. Accessed: 13 October 2023.

Android Developers 2023. Bluetooth Low Energy. URL: https://developer.android.com/develop/connectivity/bluetooth/ble/ble-overview. Accessed: 1 November 2023.

Beaufort, F. 12 August 2020. Read from and write to a serial port. Published by Google Chrome developers. URL: https://developer.chrome.com/en/articles/serial/. Accessed: 13 October 2023.

Bluetooth SIG 2023a. Bluetooth Low Energy (LE). URL: https://www.bluetooth.com/learn-about-bluetooth/tech-overview/. Accessed: 30 October 2023.

Bluetooth SIG 2023b. The Bluetooth Low Energy Primer. URL: https://www.bluetooth.com/wp-content/uploads/2022/05/The-Bluetooth-LE-Primer-V1.1.0.pdf. Accessed: 31 October 2023.

Bluetooth SIG 2023c. Part B: Link Layer Specification. URL: https://www.bluetooth.com/wp-content/uploads/Files/Specification/HTML/Core_Specification/out/en/low-energy-controller/link-layer-specification.html. Accessed: 1 November 2023.

Bluetooth SIG 2023d. The story behind how Bluetooth technology got its name. URL: https://www.bluetooth.com/about-us/bluetooth-origin/. Accessed: 3 November 2023.

Bluetooth SIG 2023e. About Us: Vision and Mission. URL: https://www.bluetooth.com/about-us/vision/. Accessed: 3 November 2023.

Bluetooth SIG 2023f. 2023 Bluetooth Market Update. URL: https://www.bluetooth.com/2023-market-update/. Accessed: 22 November 2023.

Can I Use 2023. Web Serial API. URL: https://caniuse.com/web-serial. Accessed: 18 October 2023.

Core Specification Working Group 2023. Bluetooth Core Specification. Bluetooth SIG Proprietary. URL: https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=556599. Accessed: 3 November 2023.

Eisenman, B. 2017. Learning React Native: Building mobile applications with Javascript. 2nd ed. O'Reilly. Sebastopol.

Laird 2015. Using Virtual Serial Port Service (vSP) with smart BASIC. BL 600 Development Kit application note. URL: http://cdn.lairdtech.com/home/brandworld/files/Application%20Note%20-%20Using%20VSP%20with%20smartBASIC.pdf. Accessed: 22 November 2023.

Master Table of Contents & Compliance Requirements. BLUETOOTH SPECIFICATION Version 4.0. URL: https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=456433. Accessed: 31 October 2023.

MDN 2023a. Window: postMessage() method. URL: https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage. Accessed: 10 November 2023.

MDN 2023b. Base64. URL: https://developer.mozilla.org/en-US/docs/Glossary/Base64. Accessed: 22 November 2023.

Meta Open Source 2023. useRef. URL: https://react.dev/reference/react/useRef. Accessed: 10 November 2023.

Moilanen, T., Ojasalo, K. & Ritalahti, J. 2022. Methods for development work: New kinds of competencies in business operations. Books on Demand GmbH. Helsinki.

PalmSens 2023. PalmSens. URL: https://www.palmsens.com/app/uploads/2021/12/emstat-go-portable-potentiostat-for-oem-palmsens.jpg.webp. Accessed: 29 December 2023.

Scrum.org 2023. What is Scrum? Scrum.org resources. URL: https://www.scrum.org/resources/what-scrum-module. Accessed: 15 November 2023.

Teknologiateollisuus 2023. Healthtech industry in Finland. Healthect blog post. URL: https://healthtech.teknologiateollisuus.fi/en/healthtech-industry-finland#:~:text=In%202022%20the%20value%20of,16%20billion%20euros%20trade%20surplus. Accessed: 22 November 2023.

Texas Instruments 2016a. Generic Access Profile (GAP). URL: https://software-dl.ti.com/lprf/simplelink_cc2640r2_sdk/1.35.00.33/exports/docs/ble5stack/ble_user_guide/html/ble-stack/gap.html. Accessed: 31 October 2023.

Texas Instruments 2016b. Host Controller Interface. URL: https://software-dl.ti.com/lprf/sim-plelink_cc2640r2_sdk/1.35.00.33/exports/docs/ble5stack/ble_user_guide/html/ble-stack/hci.html. Accessed: 2 November 2023.

Texas Instruments 2016c. Logical Link Control and Adaptation Layer Protocol (L2CAP). URL: https://software-dl.ti.com/lprf/simplelink_cc2640r2_sdk/1.35.00.33/ex-ports/docs/ble5stack/ble_user_guide/html/ble-stack/l2cap.html#logical-link-control-and-adaptation-layer-protocol-l2cap. Accessed: 3 November 2023.

Woolley, M. 10 August 2016. A Developer's Guide to Bluetooth Technology. Bluetooth.com blog. URL: https://www.bluetooth.com/blog/a-developers-guide-to-bluetooth/. Accessed: 1 November 2023.

Appendices

Appendix 1. Code example of ReactJS sending messages to React Native

```
/**
 * Post messages to window with native postMessage().
 * https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage
 *
 * To post messages to React Native, call postMessageToRN-function.
 */
window.message = (message: MessagePayload): void => {
  const {
    type,
    payload,
    callback,
    id: requestId,
    endOfResponseCharacter,
  } = message;

  // Post message to React Native.
  if (window.ReactNativeWebView) {
    window.ReactNativeWebView.postMessage(
      JSON.stringify({ type, payload, id: requestId, endOfResponseCharacter })
    );
  }

  // window for IOS, document for Android.
  const globalObjectToUse = navigator.userAgent.includes('iPhone')
    ? window
    : document;

  // Add listener specifically for this message.
  globalObjectToUse.addEventListener(
    'message',
    function cb(event: MessageEvent) {
      const message = event.data;
      // TODO check types.
      const { payload: response } = JSON.parse(message);

      if (callback) {
        // Check that response is meant for this listener.
        if (requestId === response.id) {
          // React Natives serial port description decides
          // when measurement is done.
          if (response.done) {
            globalObjectToUse.removeEventListener('message', cb);
          }

          callback(response);
        }
      }
    }
  );
};
```

Appendix 2. Code example of React Native managing messages with web application

```
import React, { useRef, useCallback, useEffect } from 'react';
import { WebView } from 'react-native-webview';
import { IMessage } from './../../utils/interfaces';
import { parseJsonObject } from './../../utils/common';
import { ActionTypes } from '../../utils/RNContext';
import { MessageTypes } from './../../utils/RNContext/types.d';

const WebViewUI = (): JSX.Element => {
  const webViewRef = useRef<WebView>(null);

  /**
   * Post messages to webview.
   */
  const postMessage = useCallback(
    (messageToWeb: {
      type: string;
      payload: string | object | boolean | [];
    }) => {
      const dataAsString = JSON.stringify(messageToWeb);

      webViewRef?.current?.postMessage(dataAsString);
    },
    [webViewRef],
  );

  /**
   * Triggered when device bluetooth device listener updates onRead-value.
   */
  useEffect(() => {
    if (onRead?.data) {
      postMessage({
        type: MessageTypes.SEND_MESSAGE,
        payload: onRead,
      });

      actions.dispatch({ type: ActionTypes.CLEAR_ON_READ });
    }
  }, [postMessage, actions, onRead]);

  /**
   * Middleware to detect incoming messages.
   */
  const onMessageService = (event: React.ChangeEvent) => {
    const { data }: any = event.nativeEvent;
    const { type, payload, id, endOfResponseCharacter } = parseJsonObject(
      data,
    ) as IMessage;

    // Do something wih the message.
  };

  return (
    <WebView
      ref={webViewRef}
      source={{
        uri: 'https://your-webservice.com/',
      }}
      onMessage={onMessageService}
    />
  );
};

export default WebViewUI;
```

Appendix 3. BLE PLX Bluetooth manager in React Native with React Context

```
import React, {
  createContext,
  useContext,
  useEffect,
  useReducer,
  useCallback,
  useState,
  useRef,
} from 'react';
import { Text } from 'react-native';
import {
  BleError,
  BleManager,
  Characteristic,
  Device,
} from 'react-native-ble-plx';
import base64 from 'react-native-base64';
import Geolocation from 'react-native-geolocation-service';
import { useTranslation } from 'react-i18next';
import reducer from './actions';
import {
  IAppState,
  IContextAction,
  ActionTypes,
  BluetoothEventSubscription,
  BluetoothStatuses,
  IWrite,
  MessageTypes,
} from './types.d';
import { EndOfResponseCharacter } from './../interfaces';
import {
  requestBluetoothPermissions,
  getCommandsAsBase64Chunks,
} from './../common';

// Peripheral specific values.
const PICO_SERVICE_UUID = '00000000-0000-0000-0000-000000000000';
const READ_CHARACTERISTIC_UUID = '00000000-0000-0000-0000-000000000000';
const WRITE_CHARACTERISTIC_UUID = '00000000-0000-0000-0000-000000000000';
const MAX_MTU = 23; // maximum transmission unit
/**
 * Abort command is 'Z\n', but abort's response is 'Z+' or 'Z\n+\n'.
 * Seems to be more safe to check only the character 'Z' and assume it is now a reserved character.
 */
const ABORT_COMMAND = 'Z\n';

interface IAppContext {
  state: IAppState;
  actions: {
    dispatch: React.Dispatch<IContextAction>;
    write({ command, id, endOfResponseCharacter }: IWrite): void;
    connectDevice(
      deviceToConnect: Device,
    ): Promise<void | unknown | undefined> | void;
    cancelDeviceConnection(): Promise<void> | void;
    scan(): void;
  };
}

const initialState: IAppState = {
  device: null,
  bluetoothStatus: BluetoothStatuses.IDLE,
  discovering: false,
  scannedDevices: [],
  onRead: null,
  isLoading: false,
  isInitialLoad: true,
  showWebView: true,
  deviceStatus: {
    hasBluetoothPermission: false,
    isBluetoothOn: false,
    isLocationServicesEnabled: false,
  },
  error: null,
  isDemoMode: false,
};

const RNContext = createContext<IAppContext>({
  state: initialState,
  actions: {
    dispatch: () => {},
    write: () => {},
    connectDevice: () => {},
    cancelDeviceConnection: () => {},
    scan: () => {},
  },
});

const useRNContext = () => useContext(RNContext);

type P = {
  children: React.ReactNode;
};

const manager = new BleManager();

/**
 * React Native's state management.
 * When declared on the top level, use any property passed to provider
 * by importing useRNContext and using it:
 * const { state, actions } = useRNContext().
 *
 */
const RNContextProvider = ({ children }: P): JSX.Element => {
  const [state, dispatch] = useReducer(reducer, initialState as IAppState);
  // isAbortActive is just reassuring the response character 'Z' in case the serial reader would receive it in some other
event than abort.
  const isAbortActive = useRef(false);

  const { t } = useTranslation();
  // One subscription must be used at a time, as the Potentiostat does not know
  // to which request the response belongs to.
  const [queue, setQueue] = useState<IWrite[]>([]);
  const [subscriptions, setSubscriptions] = useState<{
    // bluetoothModule: BluetoothEventSubscription;
    // bluetoothSocket: BluetoothEventSubscription;
    reader: BluetoothEventSubscription | null;
  }>({
    reader: null,
  });

  const { device, bluetoothStatus, isLoading } = state;

  /**
   * Subscriber callback listening to read-events.
   *
   * Sets response to state as onRead.
   */
  const _writeCallback = useCallback(
    (
      error: BleError | null,
      characteristic: Characteristic | null,
      id: string | number,
      endOfResponseCharacter: EndOfResponseCharacter = '\n',
    ) => {
      try {
        // Value contains data.
        if (!characteristic?.value) {
          throw new Error('Characteristic or value is null');
        }

        const data = base64.decode(characteristic.value);
        const isAbortDone =
          isAbortActive.current === true && data.startsWith('Z');

        console.log('[READ] ', data);

        /**
         * Based on the documentation, a MethodScript-measurement contains the character '*',
         * and other commands contain '\n' when the reading is done.
         *
         * https://www.palmsens.com/app/uploads/2020/04/Emstat-Pico-communication-protocol-V1.2.pdf.
         */
        const done = data.includes(endOfResponseCharacter);

        dispatch({
          type: ActionTypes.SET_ON_READ,
          payload: {
            data: data,
            timestamp: new Date(), // Add the current date
            id: id,
            done,
          },
        });

        if (done || isAbortDone) {
          dispatch({ type: ActionTypes.SET_IS_LOADING, payload: false });

          if (isAbortDone) {
            isAbortActive.current = false;
          }
        }
      } catch (e) {
        // ignore.
      }
    },
    [],
  );

  /**
   * Adds a continuous observer to device.
   */
  const addSubscription = useCallback(
    async (
      id: string | number,
      endOfResponseCharacter?: EndOfResponseCharacter,
    ) => {
      if (!device) {
        throw new Error('No device found');
      }
      dispatch({ type: ActionTypes.SET_IS_LOADING, payload: true });

      const subscription = device.monitorCharacteristicForService(
        PICO_SERVICE_UUID,
        READ_CHARACTERISTIC_UUID,
        (error, writeCharacteristic) =>
          _writeCallback(
            error,
            writeCharacteristic,
          id,
          endOfResponseCharacter,
        ),
      );

      setSubscriptions(prevState => {
        return {
          ...prevState,
          reader: subscription, // contains remove().
        };
      });
    },
    [_writeCallback, device],
  );

  /**
   * Maintain active connection.
   */
  const _maintainBleConnection = async (peripheral: Device) => {
    if (peripheral) {
      peripheral.monitorCharacteristicForService(
        PICO_SERVICE_UUID,
        READ_CHARACTERISTIC_UUID,
        () => null,
      );
    } else {
      console.log('[_maintainBleConnection] error: No Device Connected');
    }
  };

  /**
   * Call subscription.reader.remove to cancel any subscription in state.
   * Remove subscription from state.
   */
  const _removeSubScription = useCallback(() => {
    if (subscriptions.reader) {
      subscriptions.reader.remove();

      setSubscriptions(prevState => {
        return {
          ...prevState,
          reader: null,
        };
      });
    }
  }, [subscriptions.reader]);

  /**
   * Device disconnection callback listener.
   *
   * Set device in state to null and inform webview about the disconnection.
   */
  const _onDisconnect = useCallback(() => {
    // Send message to Web.
    dispatch({
      type: ActionTypes.SET_ERROR,
      payload: {
        type: 'error',
        title: t('RNContext.disconnect_title'),
        description: t('RNContext.disconnect_description'),
        action: {
          type: MessageTypes.SET_SHOW_WEBVIEW,
          label: t('common.reconnect'),
        },
      },
    });
  }, [t]);

  /**
   * Connect to a specified device.
   */
  const connectDevice = useCallback(
    async (peripheral: Device) => {
      try {
        console.log('CONNECT DEVICE');

        const deviceConnection = await manager.connectToDevice(peripheral.id);

        dispatch({ type: ActionTypes.SET_DEVICE, payload: deviceConnection });

        await deviceConnection.discoverAllServicesAndCharacteristics();

        manager.stopDeviceScan();

        dispatch({ type: ActionTypes.SET_DISCOVERING, payload: false });

        _maintainBleConnection(deviceConnection);

        // Add listener to detect disconnection.
        const subscription = manager.onDeviceDisconnected(
          deviceConnection.id,
          () => {
            _onDisconnect();
            subscription.remove();
          },
        );
      } catch (error) {
        console.error('[connectDevice] error', error);
      }
    },
    [_onDisconnect],
  );

  /**
   * Write command to Pico-devices write-characteristic.
   *
   * In BLE communication the command's format must be base64.
   * Default MTU is 23 bytes. Therefore, the long commands, such as MethodScripts,
   * must be split down and written in chunks.
   */
  const write = useCallback(
    async ({ command, id, endOfResponseCharacter }: IWrite) => {
      if (!device) {
        return;
      }

      if (command === ABORT_COMMAND) {
        isAbortActive.current = true;
        _removeSubScription();
      }

      if (!subscriptions.reader) {
        addSubscription(id, endOfResponseCharacter);
      } else if (subscriptions.reader && isLoading) {
        // Has active writing in progress, add to queue.
        setQueue([...queue, { command, id, endOfResponseCharacter }]);

        return;
      }

      const chunks = getCommandsAsBase64Chunks(command, MAX_MTU);

      // Write all chunks to characteristic.
      if (chunks) {
        for (const chunk of chunks) {
          await device.writeCharacteristicWithResponseForService(
            PICO_SERVICE_UUID,
            WRITE_CHARACTERISTIC_UUID,
            chunk,
          );
        }
      }
    },
    [
      addSubscription,
      device,
      subscriptions.reader,
      queue,
      isLoading,
      _removeSubScription,
    ],
  );

  const cancelDeviceConnection = async (): Promise<void> => {
    if (device) {
      await manager.cancelDeviceConnection(device.id);
    }
  };

  const _isDuplicateDevice = (
    devices: Device[],
    nextDevice: Device,
  ): boolean => {
    return devices.findIndex(d => nextDevice.id === d.id) > -1;
  };

  /**
   * Scan for all advertising bluetooth devices.
   *
   * Devices filtered by both service UUID and device identifier.
   */
  const scan = useCallback((serviceUuid: string | null = null): void => {
    let foundDevices: Device[] = [];

    dispatch({ type: ActionTypes.SET_DISCOVERING, payload: true });

    manager.startDeviceScan(
      serviceUuid ? [serviceUuid] : null,
      null,
      async (error, advertisingDevice) => {
        if (error) {
          // Handle error (scanning will be stopped automatically)
          console.error('Error: ', error);

          return null;
        }
        // Connects to first found PS-device.
        // TODO replace with Fepod hardware identifier when there is one.
        if (advertisingDevice) {
          if (!_isDuplicateDevice(foundDevices, advertisingDevice)) {
            foundDevices.push(advertisingDevice);

            if (foundDevices.length > 0) {
              dispatch({
                type: ActionTypes.SET_SCANNED_DEVICES,
                payload: foundDevices,
              });
            }
          }
        }
      },
    );
  }, []);

  /**
   * Check if one location request succeeds.
   * When succeeding, we can assume location is on. If error is thrown, location is off.
   * This is separate from granting access to use location services in the app.
   */
  const _setIsLocationServicesOn = (): void => {
    Geolocation.getCurrentPosition(
      position => {
        if (position) {
          dispatch({
            type: ActionTypes.SET_DEVICE_STATUS,
            payload: {
              isLocationServicesEnabled: true,
            },
          });
        }
      },
      error => {
        // See error code charts below.
        console.log(error.code, error.message);
      },
      { enableHighAccuracy: true, timeout: 15000, maximumAge: 10000 },
    );
  };

  /**
   * If connected successfully, handle UI flow.
   */
  useEffect(() => {
    if (bluetoothStatus === BluetoothStatuses.CONNECTED) {
      // Hack to give time for Success message to be displayed.
      setTimeout(() => {
        dispatch({
          type: ActionTypes.SET_BLUETOOTH_STATUS,
          payload: BluetoothStatuses.IDLE,
        });
      }, 500);

      setTimeout(() => {
        dispatch({ type: ActionTypes.SET_SHOW_WEBVIEW, payload: true });
      }, 500);
    }
  }, [bluetoothStatus]);

  /**
   * Ask for permissions and check bluetooth initial state.
   */
  useEffect(() => {
    if (!device) {
      dispatch({
        type: ActionTypes.SET_BLUETOOTH_STATUS,
        payload: BluetoothStatuses.CONNECTING,
      });

      // Check user permissions.
      requestBluetoothPermissions(async (isGranted: boolean) => {
        // TODO decide if subscription should exist forever.
        const subscription = manager.onStateChange(bluetoothState => {
          const btEnabled = bluetoothState === 'PoweredOn';

          if (isGranted) {
            // Check if location services are on.
            _setIsLocationServicesOn();
          }

          if (isGranted && btEnabled) {
            scan();
          }

          subscription.remove();

          dispatch({
            type: ActionTypes.SET_DEVICE_STATUS,
            payload: {
              hasBluetoothPermission: isGranted,
              isBluetoothOn: btEnabled,
            },
          });
        }, true);
      });
    }
  }, [device, scan]);

  /**
   * Cleanup subscription.
   */
  useEffect(() => {
    if (!isLoading && subscriptions.reader) {
      _removeSubScription();
    }
  }, [isLoading, subscriptions.reader, _removeSubScription]);

  /**
   * Call write-commands from queue.
   */
  useEffect(() => {
    if (!subscriptions.reader && queue.length > 0) {
      write(queue[0]);

      // Remove queue item.
      setQueue(queue.slice(1));
    }
  }, [subscriptions.reader, queue, write]);

  if (!state.scannedDevices || state.scannedDevices.length === 0) {
    <Text>{t('common.loading')}</Text>;
  }

  return (
    <RNContext.Provider
      value={{
        state,
        actions: {
          dispatch,
          connectDevice,
          cancelDeviceConnection,
          write,
          scan,
        },
      }}>
      {children}
    </RNContext.Provider>
  );
};

export { RNContextProvider, useRNContext, ActionTypes, BluetoothStatuses };
```

Appendix 4. Example of Base64 encoding and command splitting

```
/**
 * Split commands to smaller chunks, as BLE communication allows 23 bytes by default.
 *
 * @returns {string[]} Commands in base64 format, split to be <= MTU.
 * (Maximum transmission unit)
 */
export const getCommandsAsBase64Chunks = (
  command: string,
  mtu: number,
): string[] => {
  if (!command) {
    return [];
  }

  // Convert each row to base64.
  const base64Chunks = [command].map(row => {
    return base64.encode(row);
  });

  // Split rows into chunks that are <= than specified byte size.
  return base64Chunks.reduce((acc: string[], res: string) => {
    const maxSizedCommand: RegExpMatchArray | null = res.match(
      new RegExp(`.{1,${mtu}}`, 'g'),
    );

    if (maxSizedCommand) {
      return [...acc, ...maxSizedCommand];
    }

    return acc;
  }, []);
};
```

Appendix 5. Interview Questions

- Did the project provide a solution to a problem Fepod faced?
- What were the project’s key objectives that were crucial for Fepod’s business?
- How were the key objectives met?
- Has the project had an impact in the industry or how has it been recognized?
- What where the key factors for project to be successful? In case the project was unsuccessful, key factors why?
- Was the development process done following agile development processes? How did that work out?
- What feedback has Fepod received from stakeholders and their clients?
- Did the project have a positive impact in your business? How has it been visible?
- Are there any lessons learned/insight gained/further development plans for the PoC?
- Did Fepod have any other options for software, than building their own?
- Anything else worth mentioning?