ulm university universität uulm

Ulm University | 89069 Ulm | Germany

Medical Faculty
Institute of Medical
Systems Biology

Enhancing Mobile Data Collection Applications with Sensing Capabilities

Master's thesis at Ulm University

Submitted by:
Robin Martin
robin.martin@uni-ulm.de

Reviewer:
Prof. Dr. Hans A. Kestler
Prof. Dr. Rüdiger Pryss

Supervisor:
Dr. Johannes Schobel

2020

Version from June 24, 2020

© 2020 Robin Martin

This work is licensed under the Creative Commons. Attribution-NonCommercial-ShareAlike 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/de/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Composition: PDF-LaTeX 2_ε

Abstract

Over the past years, using smart mobile devices for data collection purposes has become ubiquitous in many application domains, replacing traditional pen-and-paper based data collection approaches. However, in many cases, modern approaches only aim to replicate traditional data collection instruments (e.g., paper-based questionnaires) in a digital from (e.g., smartphone surveys). Thereby, the full potential of smart mobile devices is often not fully exploited. Most modern smart mobile devices comprise a variety of sensing capabilities, which may provide valuable data, and thus insights. In addition, external sensors and devices may be easily connected to become part of the overall data collection process. In order to integrate sensing functionality into existing data collection applications, one has to address each desired sensor manually from within the application, which may cause severe development effort. Alternatively one can fall back on dedicated sensing frameworks to perform sensing operations. However, the latter are often targeted towards one specific mobile platform (e.g., *iOS* or *Android*) or lack required functionality, which may also lead to unnecessary development overhead when implementing mobile data collection applications. To cope with these issues, a cross-platform mobile sensing framework that can be used within large-scale mobile data collection scenarios was developed in the context of this thesis. Thereby, an in-depth look at existing mobile sensing frameworks as well as common use case scenarios is taken. Further, requirements derived from the latter are explicitly stated and were taken into consideration in the course of the overall development process. The latter is documented and discussed in detail in the course of this thesis, including the design of a framework architecture, implementation details and the integration of the framework into mobile data collection applications.

Acknowledgment

At this point, I would like to thank my family and friends who supported and continuously motivated me throughout my studies and during the preparation of this thesis.

Most notably, I would like to thank my supervisor Dr. Johannes Schobel for his excellent guidance, mentorship and support over the course of my master studies, throughout this thesis and beyond.

Contents

1 Introduction **1**
1.1 Outline 2

2 Fundamentals **5**
2.1 Bluetooth Low Energy 5
2.2 Cross-Platform Development 10
2.3 Web Components 13

3 Evaluation of Existing Mobile Sensing Frameworks **15**
3.1 SensingKit 15
3.2 Event-based Sensor Framework 17
3.3 Google Fit 20
3.4 Comparison 23

4 Application Scenarios **29**
4.1 Remote Patient Monitoring 29
4.2 Intensive Longitudinal Methods 31

5 Towards a Generic Sensor Framework **33**
5.1 Requirements 34
5.2 Framework Architecture 37

6 Implementation **41**
6.1 Technologies 41
6.2 Capacitor Plugin Implementations 42
6.3 Software Architecture 48
6.4 Sensor Implementations 57

7 Enhancing Mobile Applications with Sensing Capabilities **67**
7.1 Framework Setup within Application 67
7.2 Addressing Sensors 69

7.3 Extending the Framework with Custom Sensors 72
7.4 Conclusion 73

8 Summary **75**
8.1 Outlook 77

A Sources **85**
A.1 Implementation of Geolocation Component 85
A.2 Implementation of Custom Battery Sensor 88

1 Introduction

Over the past decade, smart mobile devices, such as smartphones or tablet computers, became an ubiquitous part of people's everyday life. Due to their characteristic properties (i.e., being portable, sensor rich, programmable and powerful computing devices) using them as a tool for data collection purposes is becoming increasingly popular in many application domains. Compared to traditional, pen-and-paper based data collection, digital data collection approaches offer various benefits.

First of all, the unnecessary burden on environment, for example by printing thousands of sheets of paper, may be lowered drastically by relying on digital solutions. Further, digital approaches can minimize the overall cost of data collection, and thus, allow researchers to conduct studies with large sample sizes (e.g., clinical trials) with ease [1, 2]. Also, collected data may be stored and processed immediately and must not be digitized manually in tedious and error prone transcription tasks. The latter contributes to an increase in overall data quality significantly [2].

However, many of the existing tools for mobile data collection (e.g. survey configuration tools) exclusively rely on a form based approach in order to reproduce traditional paper-based questionnaires in a digital way on smart mobile devices. Thereby, a lot of additional benefits that may be gained through the usage of smart mobile devices for data collection are neglected. Most importantly, smart mobile devices themselves comprise a variety of internal sensing capabilities (e.g., accelerometer, *GPS* or microphones). The latter may be addressed in order to gather rich contextual information in addition to data collected through regular forms. Further, smart mobile devices offer various wired (e.g., *USB*) as well as wireless (e.g., *Bluetooth*, *WiFi*) connectivity options, and thus, allow for the integration of external sensors and devices into the data collection process [3].

Nevertheless, enhancing mobile data collection applications to also collect data from internal as well as external sensors can be a challenging task. Most existing mobile data collection tools (e.g., survey configurators) do not provide the opportunity for configuring data collection instruments that also gather sensor information. Hence, one has to rely on dedicated, custom implemented data collection applications, rather than general solutions, for addressing sensors during the data collection process. However, implementing dedicated mobile applications can be both, cost- and time-intensive as it requires sophisticated knowledge about platform-specific ways of accessing sensors as well as the sensors themselves, which may differ greatly in their characteristics. Also, most of the time, dedicated solutions only serve one specific use case and may be superfluous afterwards. While approaches for generically addressing sensors on mobile devices exist, they are often limited to a specific mobile platform and restricted to a small set of available sensors. In order to cope with these issues, the aim of this thesis is to design and implement a generalized mobile sensing framework, which may be integrated into existing data collection tools and applications. Using this framework, one should be able to build data collection instruments and applications which gather sensor data from a variety of different sensors on multiple platforms with ease.

1.1 Outline

To begin with, Chapter 2 gives an overview over fundamental aspects that might be required for further understanding in later parts of the thesis. Thereby, covered topics include the *Bluetooth Low Energy* standard (Section 2.1), *Cross-Platform Development* (Section 2.2) as well as a brief introduction to *Web Components* (Section 2.3). Following, Chapter 3 is concerned with different existing mobile sensing frameworks. The latter are presented in detail and evaluated from various points of view. Next, Chapter 4 presents multiple use case scenarios, where gathering data from mobile sensors could find beneficial appliance, with particular attention to *Remote Patient Monitoring* and *Intensive Longitudinal Methods*. With insights from previous chapters in mind, a set of requirements, the sensing framework to be developed has to fulfill, are elaborated in Chapter 5. Further, according to elaborated requirements, a general architecture

for the sensing framework is defined. Chapter 6 then covers in-depth implementation details about the developed framework, including utilized technologies (Section 6.1), custom plugin implementations (Section 6.2), details about the software architecture (Section 6.3) and the actual sensor implementations within the framework (Section 6.4). In order to demonstrate how to integrate the developed framework into existing mobile application, Chapter 7 gives a closer look at the framework integration process. Thereby, the initial framework setup (Section 7.1), different ways of accessing sensor data from within the application (Section 7.2) and framework extension approaches (Section 7.3) are discussed in detail. Chapter 8, then recapitulates and discusses several aspects of the developed framework. Finally, Section 8.1 gives an outlook on how the framework, developed in the course of this thesis, could be extended with additional features in further iterations.

2

Fundamentals

In this chapter, general aspects which may be important for further understanding parts of this thesis, are introduced. Section 2.1 covers the *Bluetooth Low Energy* standard and gives a brief description about different parts of the *Bluetooth Low Energy* protocol stack. Further, *cross-platform development*, an alternative approach to developing native mobile applications, is introduced in Section 2.2.

2.1 Bluetooth Low Energy

Bluetooth Low Energy (BLE) is a wireless technology for short-range communication developed and maintained by the *Bluetooth Special Interest Group* (SIG). In 2010, BLE became part of the *Bluetooth 4.0* core specification. As the name implies, one of the key advantages of BLE over previous *Bluetooth* implementations is its relatively low energy consumption [5, 4]. Similar to the classic *Bluetooth* stack, the BLE protocol stack consists of two major parts, the *Controller* and the *Host* (see Figure 2.1). The layers within the controller part enable a standard interoperable wireless communication and are responsible for packet transmission and scheduling. *Host* layers, on the other hand, implement multiple network and transport protocols, which allow applications for a standard and interoperable way of communicating with peer devices [6]. As this thesis does not require in depth knowledge about BLE low-level functionality, this section focuses on the top-most three layers of the BLE protocol stack. In detail, the following sections present the *Attribute Protocol*, *Generic Attribute Profile* as well as the *Generic Access Profile*. Thereby, most of the information provided is consulted from the *Bluetooth* core specification [7].

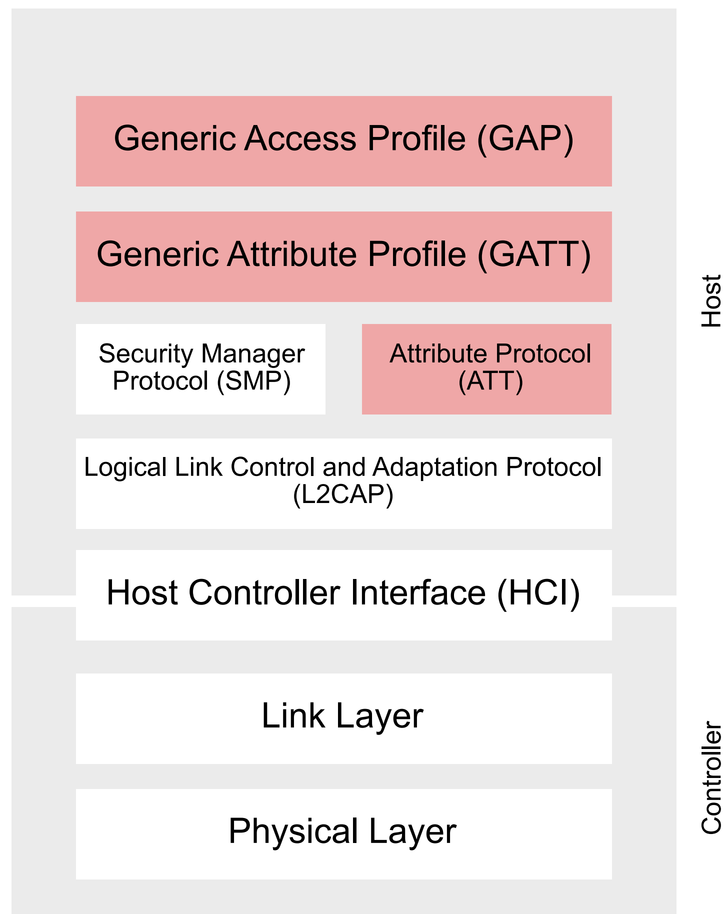


Figure 2.1: High-level overview of the *Bluetooth Low Energy* protocol stack [4]

Attribute Protocol

The *Attribute Protocol* defines the communication between two connected BLE devices. Thereby, one device takes on a server role, whereas the other one acts as a client. As the name of the protocol implies, the communication between client and server is based on the exchange of certain attributes. Attributes are defined as discrete values described by a universal unique identifier (UUID) as well as a dedicated handle. Further, attributes

may have a set of permissions associated, which allow for a more fine-grained definition of access rights (e.g., read, write or both) on given attributes. The device with the server role typically is in charge of maintaining a set of attributes, whereas the client may read or write these attributes in a typical request/response scheme. Also, the server can send unsolicited messages to the client via notifications and indications. While indications require the client to confirm the receipt of the message, no acknowledgement of receipt from the client is required when sending notification messages. This communication pattern further contributes to a more energy efficient message exchange.

Generic Attribute Profile

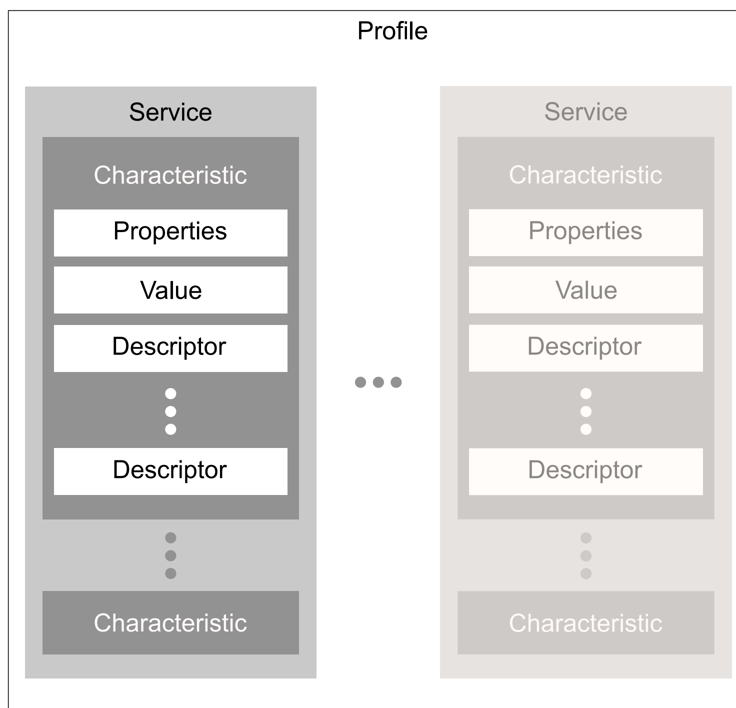


Figure 2.2: GATT Profile hierarchy [7]

Built on top of the *Attribute Protocol*, the *Generic Attribute Profile* aims to define a service framework establishing precisely how data is exchanged between two connected BLE devices on a higher level. Derived from the *Attribute Protocol*, one of the participating

devices must take a server role, while the other one must take the complementing client role. However, these roles must not be fixed for each device. The roles of participating devices are determined whenever a procedure is initiated and are released afterwards. Said procedures (e.g., service discovery, reading, writing or notifying), along with specific formats, of services and their respective characteristics, are defined within the *Generic Attribute Profile*. The latter acts as a reference frame for all *GATT* based profiles. As shown in Figure 2.2, *GATT* follows a hierarchical approach when it comes to organizing data. The hierarchy can be subdivided into three main levels with descending hierarchy and ascending granularity, as described below :

Profile. *GATT* profiles reside at the highest level. Within a profile, the structure in which data is exchanged is specified. A profile, therefore, explicitly defines basic attributes (i.e. services and characteristics) necessary in order for a device implementing the profile to work appropriately. Hence, dedicated profiles can be seen as the backbone of interoperability between devices from different manufacturers.

Service. *GATT* services can be seen as a collection of data and associated behaviors needed to accomplish a certain function or feature within a specific application scenario. A respective service definition describes all building blocks that are necessary to fulfill aforementioned function or feature. This may include mandatory and optional characteristics as well as references to other services.

Characteristic. At the lowest level of the hierarchy, *GATT* characteristics encapsulate raw data values. Alongside the actual value, which may be a single data point or an array of associated data points, a characteristic may contain additional information. The latter may be information on how the value can be accessed by the client (e.g., characteristic properties) as well as semantic or descriptive information about the underlying value (e.g., descriptors). A respective characteristic definition contains a declaration, characteristic properties and the actual value including information on how it is composed. Finally, the characteristic definition may contain a set of descriptors, which can provide further information about the value itself (e.g., descriptive texts) or allow for the configuration of the server (e.g., enabling notifications via *Client Characteristic Configuration Descriptor*).

Generic Access Profile

At the top-most level of the BLE stack, the *Generic Access Profile* defines different roles, modes and generic procedures related to the discovery of devices and services, connection management and the use of certain security levels. The four roles within the *Generic Access Profile* are defined as follows:

Broadcaster. A device in the broadcaster role continuously sends data packets over dedicated advertising channels. Thereby, communication is unidirectional and does not require connection establishment between two devices.

Observer. Complementing with devices having a broadcaster role, devices in the observer role continuously scan for broadcasters nearby. Data packets sent by broadcasters may be consumed by a corresponding observer without establishing an active connection between devices.

Peripheral. Similar to the broadcaster role, devices taking on a peripheral role initially broadcast advertising packets to their surroundings. Those packets may contain valuable information regarding the actual data the peripheral provides, for example explicit service UUIDs. In order to request data from a peripheral device, an active connection is required. However, one peripheral can only have a single connection to another device in a central role.

Central. The central role acts as a complement for the peripheral role. Centrals listen for advertising packets published by peripherals nearby. Once a packet of interest is received, a device in central role is in charge of initiating connection establishment with the publishing peripheral. Finally, after creating a successful connection, a central can interact with a corresponding peripheral as described in previous sections. As opposed to the peripheral role, devices in the central role are able to connect and manage connections to multiple peripherals.

However, devices are not limited to one of the above roles. A single device may support various roles, but can only take on one of them at a given time.

Application profiles, as described in Section 2.1, which allow for the reuse of common functionality for certain types of applications and enable interoperability between devices

from different vendors, can be built on top of the *Generic Access Profile*. Those application specific profiles are specified and maintained by members of the *Bluetooth SIG* [8, 9, 10].

In the following course of this thesis, more detailed insights into different application specific profiles are given.

2.2 Cross-Platform Development

Over the past decade, smart mobile devices have become an ubiquitous part of people's everyday life. This widespread dissemination opened the market for mobile applications running on aforementioned devices. However, there is no universal operating system for mobile devices. Nowadays, the market share of mobile operating systems is mostly split between *iOS* and *Android* [11].

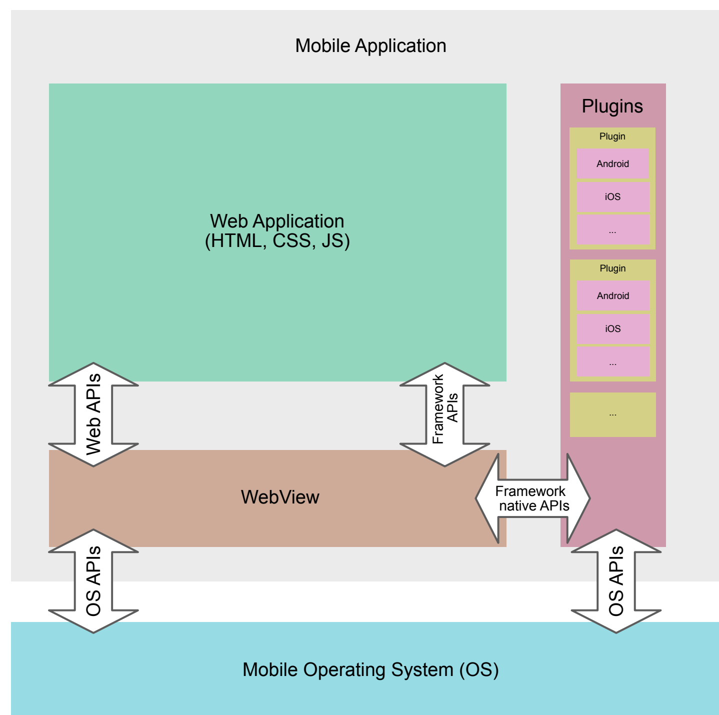


Figure 2.3: Typical software architecture in hybrid mobile applications [12]

As a result, developing mobile applications can be a quite tedious task, especially when targeting audiences from both, the *iOS* and the *Android* platform. When developing native applications for each respective platform, the latter impose the use of dedicated programming languages (e.g., *Java / Kotlin* for *Android*, *Objective C / Swift* for *iOS*), patterns and paradigms as well as platform-specific application programming interfaces upon developers.

In order to encounter the complexity of developing separate native applications, emerging cross-platform development technologies, tools and frameworks aim to pursue an alternative development approach. They allow for the creation of mobile applications for different platforms from a single code base [13]. The majority of those frameworks and tools rely on state-of-the-art web technologies in order to pursue a cross-platform approach. As opposed to native mobile applications, web-driven applications are not tied to a specific operating system to run on, but rather to a specific browser implementation of the respective platform, which is widely standardized. One of the more prominent web-driven mobile application approaches are so called *hybrid* mobile applications. The latter are assembled from three major parts [12], as can be seen in Figure 2.3.

Web Application. Hybrid mobile applications are implemented as regular web applications. Therefore, the entire business logic is written in *JavaScript*. Further, the application's user interface is defined by making use of *Hypertext Markup Language* (HTML) and *Cascading Style Sheets* (CSS). For more complex, large-scale applications, it may be advisable to fall back on common web front-end frameworks which provide a robust frame and predefined application building blocks .

Web View. The *Web View* is a platform-specific, native component (e.g. `WebView` on Android and `WKWebView` on *iOS*). It can be seen as lightweight mobile web browser which can be integrated into regular native mobile applications. As such, it provides a run-time environment for web applications within a traditional, native application.

Plugins. Since access to native resources and functionality via standard *Web APIs* may be restricted, hybrid mobile applications make use of dedicated plugins

in order to access platform-specific, native features. Thereby, a plugin consists of platform-specific code snippets, written in the native programming language of the respective platform and a dedicated *JavaScript* API, which is exposed to the application. At run-time, native code may be invoked via foreign-function interfaces by performing respective *JavaScript* API function calls.

2.2.1 Capacitor

One framework which aims to pursue the aforementioned paradigm is *Capacitor*. The self proclaimed spiritual successor of the popular *Apache Cordova* framework, provides a cross-platform application run-time for hybrid mobile applications. Therefore, it allows for developing web applications running natively on *iOS*, *Electron* and the web [14]. Access to native features is granted through dedicated *Capacitor* plugins while remaining backwards compatibility with most *Cordova* plugins [15]. For common use cases, *Capacitor* provides a set of pre-implemented native plugins, allowing access to platform-specific native features such as location tracking, file system manipulation or a mobile device's camera, to name a few. However, despite of being a relatively new framework, *Capacitor* enjoys rapidly growing community participation resulting in many custom open-source plugin implementations in order to meet specific application requirements that go beyond the already implemented plugins. To allow for the latter, *Capacitor* provides abstract plugin implementations in *Swift*, *Java* and *TypeScript*. By extending said implementations, adding custom functionality and registering them within the application at build-time, custom plugins may be accessed through common *TypeScript* plugin interfaces at run-time. One key factor separating *Capacitor* from *Cordova* is that the actual application build and publishing is not part of the framework. Rather, these tasks have to be done manually using platform-specific tools and IDEs. One could argue that this may lead to an increasing development effort, however, it also allows for a more fine grained, platform-specific configuration of resulting applications.

As *Capacitor* only provides a run-time and access to native features for hybrid mobile applications, technologies to implement the actual business logic and user interface may be selected freely by application developers. One could rely on plain *Vanilla JavaScript*,

HTML and *CSS* or fall back on popular front-end web development frameworks (e.g., *Angular*, *React*, *Vue*, etc.). The latter provide a more or less rigid frame for developing single-page and progressive web applications.

2.3 Web Components

Developing user interfaces for the web can be a quite tedious task. Implementing custom user interface controls may quickly lead to very complex markup structures. Further, scripts defining the behavior of respective elements, and associated style definitions often interfere with already existing parts of the *Document Object Model* (*DOM*) [16]. As a result, user interface definitions tend to become confusing and hard to maintain. While many web frameworks (e.g., *Angular*, *React*, etc.) aim to solve this issue by offering a component-driven approach for defining custom user interface controls, the latter can only be applied in the context of the respective web framework.

Web components aim to solve this problem by relying on already existing web APIs rather then dedicated framework APIs in order to define sophisticated, reusable user interface controls with encapsulated business logic and styles. To achieve the latter, web components use a combination of different technologies [16]:

Custom Elements: A set of web APIs allowing for the definition of custom user interface elements and their respective behavior. The latter may be directly embedded into an existing user interface.

Shadow DOM: Web APIs which allow for attaching an encapsulated *DOM* tree to an existing element and control associated behavior. The latter is rendered separate from the main document in order to keep the features of a certain element within its own scope. Hence, the behavior and style of an element can be implemented without having the risk of possible collisions with other parts of the document.

Templates: Dedicated elements (e.g., `<template>` and `<slot>`) allow for defining markup templates, which are invisible within the resulting page. Said templates may be reused as building blocks for the structure of custom elements.

The use of web components gained significant traction over the past few years, with nearly every modern browser implementing necessary APIs, required for web components to work properly. As a result, a variety of tools for supporting and speeding up the development of web components emerged.

2.3.1 Stencil

Stencil [17] is one of the tools which aims to ease the process of developing web components. In detail, *Stencil* is a compiler that generates custom, standards-compliant *HTML* elements from a set of source definitions. Generated components may integrate state-of-the-art features, such as *Virtual DOM*, asynchronous component rendering and reactive data-bindings. In order to properly define components and facilitate development (e.g., reducing boilerplate code), *Stencil* provides a set of high-level *TypeScript* utility APIs including component lifecycle-hooks and custom method and property decorators. By using the latter, the look and behavior of resulting web components may be adjusted as desired. *Stencil* components themselves are implemented using *TypeScript* to define a component's business logic, *JSX* for templating purposes and *CSS* to define the styles of a component. However, *Stencil* also offers a set of plugins, which allow for the integration of *CSS* pre-processors, such as SASS or *LESS*, into the build pipeline.

Stencil may be used to build entire web applications or dedicated web component libraries. Compiled web component outputs are self-contained, meaning they have no external dependencies and may be used in a standalone fashion. Web components built with *Stencil* can be integrated into different popular web frameworks, such as *Angular*, *React* or *Vue*, with little to no effort.

3

Evaluation of Existing Mobile Sensing Frameworks

The integration of sensors in mobile applications can be a sophisticated and tedious task for application developers. It requires in-depth knowledge about the sensors themselves, their protocols as well as platform-specific APIs, which allow to access raw sensor data. Hence, there exist mobile sensing frameworks that aim at abstracting and generalizing common sensing functionality on mobile devices. The latter allow application developers to address a wide range of sensors on a high level of abstraction, requiring minimal programming effort and knowledge about the underlying platform-specific, low-level sensor implementations.

In this chapter, a closer look at three mobile sensing frameworks is taken and they are compared and evaluated from various points of view.

3.1 SensingKit

SensingKit [18] is an open-source, multi-platform mobile sensing framework allowing to communicate with a multitude of different mobile sensors. Therefore, *SensingKit* provides dedicated client libraries for both, the *iOS* and *Android* platform. The latter may be integrated into existing native mobile applications of the respective platform. While the framework itself provides access to a wide range of device internal sensors, addressing external sensors is not supported by default. Available sensor implementations vary, depending on the underlying platform, from motion sensors (e.g., accelerometer, gyroscope), over environmental sensors (e.g., for measuring ambient light or audio levels),

up to positioning sensors (e.g., magnetometer or GPS). Currently, sensor measurements can only be retrieved by registering listeners for the respective sensor. The latter continuously publishes sensor readings to all of its listeners. Other interaction patterns, for example requesting a single measurement from a sensor, are not supported.

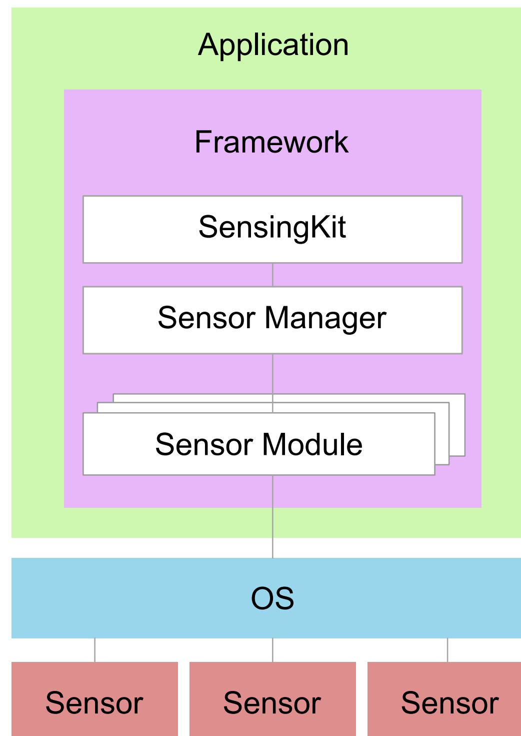


Figure 3.1: SensingKit Framework Architecture [19, 20]

The overall framework architecture can be subdivided into three main parts [19, 20], as depicted in Figure 3.1.

SensingKitLib. As the central entity of the framework, the `SensingKitLib` instance acts as an intermediary between the `SensorManager` and the mobile application. Therefore, it provides an interface, which exposes methods and functionality for registering/unregistering sensor modules and sensor listeners as well as starting/stopping sensor modules, to the mobile application. However, aforementioned actions are not performed by the `SensingKitLib` instance itself. Rather, the latter simply forwards request from the application to the `SensorManager`.

SensorManager. Located in the next lower layer, the `SensorManager` is responsible for the actual interaction with requested sensor modules. When requested, sensor modules may be instantiated dynamically by the `SensorManager`. In order to avoid duplicate instantiations of the same sensor module, the `SensorManager` keeps track of already created sensor modules. Further, application calls (e.g., for registering a sensor listener) received from higher levels, are forwarded to their corresponding sensor module in charge.

Sensor Modules. The smallest building blocks within the framework are dedicated sensor modules. Thereby, for each sensor addressable through the framework, a separate sensor module exists. Within the latter, the actual implementation logic for accessing sensor data resides. For addressing the various sensors that may be available on a certain device, the sensor modules fall back on platform-specific native APIs. In order for the `SensorManager` to be able to interact with sensor modules in a generic way, every module is derived from a common base class. The latter defines abstract functionality and behavioral blueprints, which can then be further specified in respective sub classes or within the sensor modules themselves.

This architectural approach allows for a rather easy way of adding further functionality to the framework. Due to the modular design, custom sensor implementations can be defined as their own sensor modules or inherit from existing sensor module implementations, to further specify or alter sensor behavior.

3.2 Event-based Sensor Framework

Another interesting approach towards a generic mobile sensing architecture is described in [21]. The resulting *Android*-based framework is capable of addressing a variety of different sensors. At the same time, internal as well as external sensors may be addressed via dedicated communication channels (e.g., *Bluetooth* or *USB*). The framework communication completely relies on an event-driven approach. This means that the communication with the host application as well as internal communication between

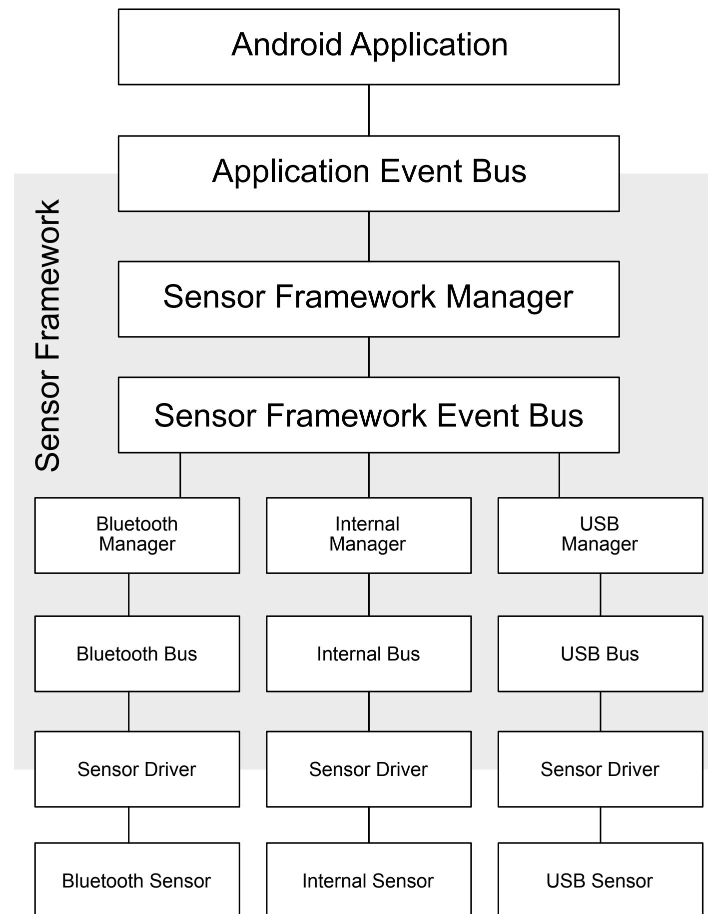


Figure 3.2: Architecture of Event-based Sensor Framework [21]

the different components of the framework takes place via dedicated events. Therefore, higher layers do not need to know about specific implementation details of layers underneath. Hence, an overall loose coupling between different parts of the framework is achieved. In order to allow for an event-driven communication approach, the framework combines multiple managing layers with dedicated event-bus layers, as demonstrated in Figure 3.2. The latter act as mediators in between respective layers.

The top-most layer of the framework is where the *Application Event Bus* resides. All communication from and to the host application is channeled through the latter. On event

receipt (e.g., for starting/stopping or requesting data from a specific sensor), the event is forwarded to the *Sensor Framework Manager*, which is responsible for the dynamic instantiation and management of the communication-protocol-specific sensor manager components (e.g., *Bluetooth Manager* or *USB Manager*). The latter, in turn, are in charge of creating instances of, managing and the configuration of requested *Sensor Drivers*. Communication with the drivers themselves takes place via respective manager-specific event buses. *Sensor Drivers* are located at the bottom of the framework hierarchy and therefore provide an interface for accessing actual hardware sensors. When receiving an event from above layers, a *Sensor Driver* must process the event correctly and address the underlying hardware sensor accordingly. Therefore, *Sensor Drivers* can be seen as the end-points for application requests to the framework.

The architecture described allows for easily extending framework functionality. Custom sensors may be integrated by implementing a dedicated *Sensor Driver*. Further, the categorization of sensors by communication channels, with dedicated protocol-specific managers allows for a modular extension of the framework in order to work with sensors supporting different communication protocols than the ones already implemented.

When it comes to the interaction with sensors from a host application, the framework supports a set of different interaction patterns, which are derived from common service interaction schemes.

Multiple-Dataresponse. After receiving a *Start*-event for a specific sensor from the host application, the sensor framework initiates a continuous stream of sample data for the given sensor. Sensor data is propagated to the application until receiving a corresponding *Stop*-event.

Single-Datarequest. When receiving a request for sensor data from the application, requested data is sent back to the latter. Thereby, requested data may be sent synchronously or asynchronously.

Recording. Recordings of data are initiated via dedicated *Start*-events. On receipt, the corresponding sensor starts gathering data internally. Finally, when the host application triggers the end of a recording through a dedicated *Stop*-event, recorded data is sent back to the host application.

Sensors within the framework must support at least one of these interaction patterns, but may also support multiple. This further contributes to the versatility of the framework, and allows application developers to address sensors according to their needs.

In addition to the actual sensing functionality of the framework, it also provides a set of feature modules, enabling event logging, serialization and deserialization of sensor data as well as visualization of gathered data. Overall, due to its versatility, the framework may be a suitable solution for a range of data collection scenarios.

3.3 Google Fit

Google Fit is a cloud-based platform for fitness and health data developed by *Google* [22]. It allows developers to gather, store and share data from a mobile device's internal sensors as well as external sensors and wearable devices centrally. Thereby, external sensors (e.g., heart rate monitors or weight scales) are integrated via *Bluetooth Low Energy*. The *Google Fit* documentation states, that the platforms offers support for all BLE sensors that implement one of the standard *GATT* application profiles (e.g., *Heart Rate Profile* or *Weight Scale Profile*) [23]. In addition to sensors supporting a standard *GATT* profile, custom sensors implementing proprietary profiles may also be integrated. The latter can be achieved within an *Android* application, by creating a dedicated service responsible for sensor interaction. Such a service for a custom sensor must inherit from the `FitnesSensorService`, which is part of the *Google Fit* package on *Android*. By adding the service to a mobile applications `manifest.xml` file, functionality of the custom sensor implementation may be exposed to *Google Fit* as a software sensor. Once the application is installed on an *Android* device, the custom sensor becomes available to be discovered and used in other applications on the same device [24]. Next to sharing certain functionality, *Google Fit* also offers the possibility to share stored data between different fitness applications and devices.

The latter is achieved through the architectural design of the *Google Fit* platform, which is depicted in Figure 3.3. The platform itself is composed out of various building blocks.

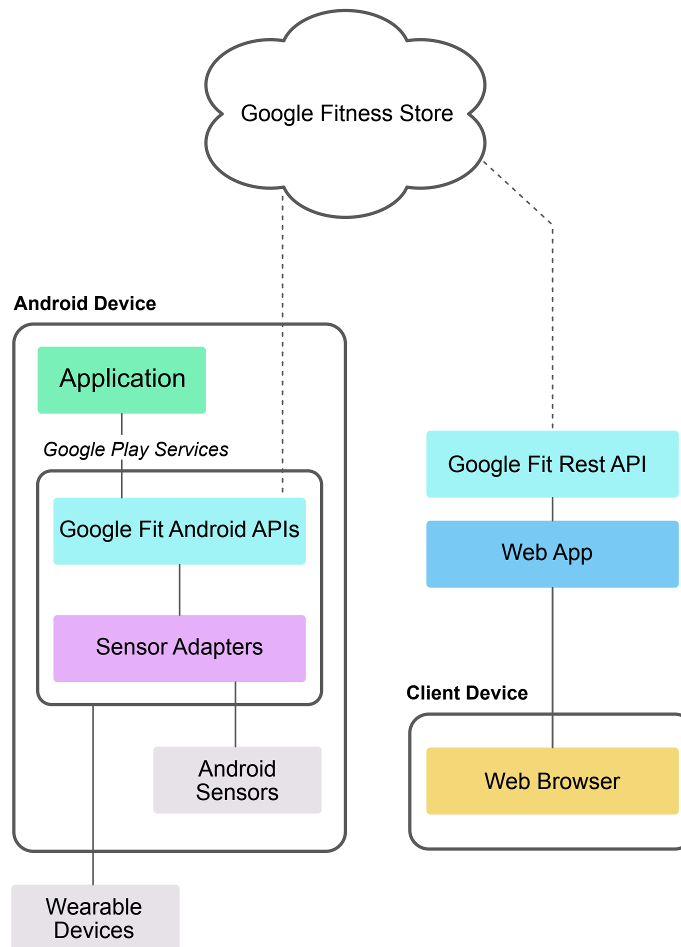


Figure 3.3: Google Fit high level architecture overview [22]

Google Fitness Store. As the central unit of the platform, the *Google Fitness Store* allows for inserting and querying sensor data gathered from different mobile devices. The store itself is a cloud based service, which can be accessed via a rich set of platform APIs.

Google Fit APIs. For interaction with the *Google Fitness Store* and locally available sensors, the platform offers various APIs. In a native *Android* environment, real-time raw sensor data may be accessed using the `Sensors API`. However,

the latter does not store sensor measurements or subscriptions to sensor data automatically. In order to persist sensor readings and subscriptions, one can fall back on the `Recording API` [25]. Further, sensor readings previously recorded and stored within the *Google Fitness Store* may be accessed using the `History API`. It can be used to perform typical CRUD operations on the *Google Fitness Store*, including bulk operations and data aggregations [26]. While those APIs are only accessible in native *Android* applications, the *Google Fit REST API* enables access to the *Google Fitness Store* for fitness applications regardless of the underlying platform.

Sensor Framework. To allow for interoperability with sensor data across different devices and platforms, high-level representations for sensors, data types, data points and activity sessions are defined within the *Sensor Framework*. Thereby, both, hardware and software sensors are defined as *Data Sources*. A *Data Source* may provide one or multiple *Data Types*. The latter provide a schema for one specific kind of fitness data (e.g., heart rate). Further, *Data Points* describe the most fine-grained entity within the framework, which represent a single reading from a *Data Source* with a specific *Data Type*. Finally, *Sessions* represent a specific time frame during which physical activity is performed by a user. *Sessions* do not include actual sensor data themselves, but provide valuable meta data to support data organization and aggregation.

Since recorded sensor data stored within the *Google Fitness Store* may be shared between different devices and applications, the *Google Fit* platform defines fine-grained permission scopes with separate access privileges. Thereby, each permission group allows for access to a set of *Data Types* residing in the latter. In order for an application to gain access to *Data Types* of a certain permission group, user consent is always required.

Similar to *Google Fit*, *Apple* provides their own framework, *HealthKit*, for storing and accessing fitness and health data [27]. However, access to said data may only be available for devices within the *Apple* ecosystem via dedicated software development kits for *iOS*, *MacOS* and *watchOS*.

3.4 Comparison

In the following, the three frameworks presented above are evaluated from several points of view. Thereby, relevant aspects include the quantity of sensors each framework is able to address, supported interaction patterns for sensors within each framework, possibilities for extending framework functionality and finally, their suitability for multi-platform and cross-platform application scenarios.

3.4.1 Available Sensors

The ability to address a variety of different sensors as well as versatility when it comes to supporting multiple communication protocols may be crucial for a sensor framework. Therefore, all of the frameworks aim to support a broad range of sensors.

SensingKit offers a set of internal default sensor implementations, which can be accessed within a mobile application. In addition to motion, environment and positioning sensors, it also allows for accessing other smartphone capabilities. The latter includes recording audio tracks, tracking the surrounding audio level or monitoring a mobile devices battery status, to name a few. While being able to address internal sensing capabilities, *Google Fit* is mostly restricted to fitness and health related, often computed software sensors used to track a users behavior or physical activity (e.g., step counter, speed, etc.). Access to low level, raw sensor data may be achieved via custom sensor implementations. As the *Event-Based Sensor Framework* aims to provide a versatile architecture for sensor integration rather then actual sensing functionality, there are no predefined implementations for commonly sought internal sensors (i.e., accelerometer, magnetometer, etc.).

When it comes to connecting external sensing devices, *Google Fit*, by default, allows for a connection with *Bluetooth Low Energy* enabled devices implementing one of the standard *GATT* profiles. Custom profiles may also be supported via custom software sensor implementations. The *Event-Based Sensor Framework* is highly versatile in supporting external sensors via different communication protocols. Due to the categorization of sensors by their connection type, with dedicated, communication-protocol-specific

sensor managers and event buses, there may be no limits for external sensor integration. However, by now, only sensor managers for connecting to external sensors via *Bluetooth* and *USB* are implemented. In contrast, *SensingKit* does not provide a default way of integrating external sensing devices, and, therefore is limited to its internal sensing capabilities.

3.4.2 Supported Interaction Patterns

The three discussed frameworks vary when it comes to different ways of interacting with provided sensors. *SensingKit*, for example, only allows for continuously monitoring sensor updates, via a common publish-subscribe-pattern, within mobile applications. In contrast, the *Event-Based Sensor Framework* offers various ways of interacting with available sensors. Depending on the sensor implementation, data may be obtained in a single request, or, similar to *SensingKit*, continuously by subscribing to value changes of a respective sensor. Further, data may be recorded over time. Thereby, recorded data is stored internally an can be returned to the application in a bulk manner, once the sensor is stopped. *Google Fit* also allows for receiving continuous sensor updates in real-time. Further, sensor readings may be recorded and stored online. Via dedicated APIs, recorded data can be obtained and aggregated.

It has to be mentioned that certain interaction patterns may be implemented within a host application, by making use of other interaction patterns. For example, requesting a single sensor reading could also be achieved by subscribing for continuous sensor readings and unsubscribing after the first value is successfully obtained. However, this may impose significant development effort upon application developers.

3.4.3 Extendability

Framework extendability is a key factor when targeting a wide range of application scenarios. A versatile sensor framework should allow for the integration of additional sensors and sensor communication protocols with minimal effort.

Therefore, all of the frameworks aim to provide ways of extending framework functionality. The architecture of the *Event-Based Sensor Framework*, for example, is designed with extendability in mind. Due to the lose coupling of its internal layers, as well as the abstraction of sensors and communication protocols, additional sensors and communication protocol manager may be implemented and integrated with ease. Further, predefined abstract classes and interfaces minimize the overall development effort necessary to extend framework functionality.

Extending *SensingKit* with additional sensors can be achieved by implementing dedicated sensor modules. This may also allow for the integration of external sensors, which the framework is not supporting by default. As an example, *Bluetooth Low Energy* sensors could be integrated by implementing an abstract generic base module, which is in charge of all communication protocol specific operations (e.g., device discovery, connection establishment/release, etc.). More fine-grained sensor-specific modules could be derived from the latter and manage sensor specific tasks, such as transforming gathered data. However, due to a lack of predefined functionality, this may cause significant development effort.

Within the *Google Fit* platform, additional sensors can be integrated by implementing a custom service inheriting the predefined *FitnessSensorService*. Also data returned from the sensor may be specified by creating a custom *DataType* as described in [28]. By doing so, additional sensors can be addressed just like predefined sensors within the framework. Further, custom software sensors may be registered within *Google Fit* making them addressable by other applications on the same device. Custom *Data Types*, however, can only be used within the application that created the latter.

3.4.4 Multi-Platform & Cross-Platform Capabilities

When it comes to multi-platform and cross-platform capabilities, the presented frameworks differ greatly. To begin with, the *Event-Based Sensor Framework* is strictly limited to the *Android* platform. Hence, it may not be the right choice for multi-platform or cross-platform development scenarios. *SensingKit*, however, provides integrations for the two most popular platforms, namely *iOS* and *Android*. This way it can be used

when developing separate native applications for the two platforms. Lastly, *Google Fit* maybe offers the most promising approach for cross-platform scenarios. While a majority of the tooling offered by *Google Fit* is targeted towards the *Android* ecosystem in the form of native *Android* libraries and APIs, the platform is not limited to *Android*. By exposing interfaces for querying and writing sensor data from and to the *Google Fitness Store* via the *Google Fit REST API*, the platform allows for participation of all applications, independent of an application's underlying mobile platform. However, in order to get real-time raw sensor data on platforms other than *Android*, one still requires sophisticated knowledge about platform specific sensor APIs. Also, using the *Google Fit REST API* requires an active internet connection, which might not be suitable or even possible in many application scenarios.

3.4.5 Conclusion

After all, it can be said that the visited frameworks themselves, despite of sharing some similarities, differ widely within the four evaluation categories. While one framework may have advantages over the other ones in one category, others may do better in another category. Hence, when it comes to choosing a one specific framework to integrate into a mobile application, the decision should be made depending on aspired use cases and application scenarios. For example, when aiming to integrate fitness and health data into an application, *Google Fit* may be the right choice. In contrast, when creating applications, targeted towards *Android* and *iOS*, that require access to low level, raw sensor data (e.g., gyroscope, accelerometer, etc.) *SensingKit* could be a suitable choice. Likewise, *Android* applications requiring custom sensing capabilities with a range of internal and external sensors in a highly versatile manner, the *Event-Based Sensor Framework* would be a perfect fit.

However, where all of the frameworks seem to have downsides is when it comes to cross-platform capabilities. Mainly, framework APIs and tools are targeted towards native application development on respective platforms. Using the same tool sets in order to equip mobile applications with sensing capabilities on multiple platforms, without

requiring an active internet connection, could ease sensor integration and, therefore minimize development effort drastically.

4 Application Scenarios

While Chapter 3 presented and discussed different mobile sensing frameworks, this chapter is concerned with specific use case scenarios, where such frameworks could find appliance. Thereby, the most prominent application scenario may be the enhancement of existing mobile data collection applications with sensing capabilities. Using sensors for gathering passive as well as active data during the overall data collection process is enjoying growing popularity and may be suitable in a wide range of application domains. Therefore, selected real-world application scenarios, with special regard to the health care and clinical research domains are presented in the following sections.

4.1 Remote Patient Monitoring

According to the *World Population Ageing* report [29], the global number of elderly people increased substantially within recent years, with about 901 million people aged 60 years or above in 2015. This trend is projected to go even further, with an estimated increase of 56 % until 2030 [29]. Meanwhile, the number of people suffering from chronic diseases such as heart failures or diabetes increases at a staggering rate [30]. However, medical systems and institutions around the world are far away from being able to cope with these trends. With an increasing number of people requiring medical treatment, traditional healthcare delivery approaches (e.g., on-sight patient examinations) could easily reach their limits [30]. Therefore, a shift towards delivering remote healthcare by relying on digital approaches could potentially benefit all participants of the healthcare delivery process.

As a result, one central research topic focuses on the conception of digital systems, enabling continuous monitoring of the health status of people with medical conditions outside of clinical environments.

For instance, *Bot et al.* [31] aim to evaluate the feasibility of remotely collecting information about changes in the severity of symptoms for patients diagnosed with Parkinson disorder (PD) as well as their sensitivity to medication. Rather than traditional approaches, where affected patients have to visit a physician every 4-6 months, the approach of *Bot et al.* requires patients to participate in self-assessments on a daily basis. The latter could reveal opportunities for interventions, which might significantly increase the quality of live of affected patients. Thereby, regular self-assessments take place via the *mPower* mobile application. Using the application, participants have to fill out PD specific questionnaires, such as the *Parkinson Disease Questionnaire* (PDQ-8) on a monthly basis. Further, participants are asked to fulfill physical tasks on a day-to-day basis. The latter incorporate smartphone sensors such as microphones, for recording voice activities, or accelerometers and gyroscopes for evaluating the patients gait and balance during walking activities. Data collected through the *mPower* application,including self-reports and sensor readings, is shared with research teams for further analysis and evaluation.

A system for remotely monitoring patients with congestive heart failure (CHF), is described in [32]. *WANDA* (Weight and Activity with Blood Pressure Monitoring System) aims to facilitate the early detection of key clinical symptoms, prevention, monitoring and treatment of CHF patients. The system architecture consists of sensors for gathering CHF related data from patients and back-end technologies such as web servers and databases for data storage and analysis. While the first iteration of *WANDA* was designed for patients that are unfamiliar with smart mobile devices (e.g., elderly patients) and transferred data from sensors (e.g., weight scales, blood-pressure monitors) directly through a phone line system, a second version of *WANDA* incorporated smartphones for data collection and transfer. More precisely, the second version uses *Bluetooth*-based weight-scales and blood-pressure monitors, smartphone-internal sensors for activity monitoring and fall detection as well as a variety of symptom questionnaires which can be filled out by patients directly on the mobile device. A real-life study incorporating

WANDA showed, that by using the system, the number of weight and blood-pressure measurements that fall out of an acceptable range can be successfully reduced.

Apart from monitoring patients with chronic diseases, *Remote Patient Monitoring* approaches may also be applicable in a range of other medical scenarios. For example *Marko et al.* [33] investigated on the feasibility of using mobile applications in combination with other connected devices for monitoring patients health in prenatal care scenarios. Therefore, study participants received a mobile application, a digital weight scale and blood-pressure cuff for collecting data at home for the duration of their pregnancy. Collected data was then assessed for irregularities in weight and blood pressure to generate alerts for both, patients and clinicians. In the course of this study, the remote assessment approach demonstrated a high patient satisfaction and could help identify two episodes of abnormal weight gain.

4.2 Intensive Longitudinal Methods

The term *Intensive Longitudinal Methods* is an umbrella term to describe a variety of research methodologies such as *experience sampling*, *daily diaries* or *ecological momentary assessment* [34]. Said methodologies may be used to examine thoughts, feelings or behaviors in their natural, real-time contexts on a high frequent basis over an extensive period of time [34]. These days, intensive longitudinal studies often rely on smart mobile devices (e.g., smartphones or tablets) to gather data from research participants in their day-to-day lives. A key benefit of using smart mobile devices for collecting longitudinal data is, that in addition to active data (e.g., from self-reports), passive data (e.g., from smartphone sensors) may be collected to provide rich, contextual information [1].

For instance, the *TrackYourTinnitus* platform [35, 36] relies on ecological momentary assessments in order to support the assessment of tinnitus symptoms for researchers and affected patients. Tinnitus, a disorder leading to the perception of sound with no corresponding external source of sound, affects about 10-15 % of the world's population [35]. Since tinnitus is a highly subjective perception, assessing the symptoms

can only take place with the help of reports from affected patients. Further, in order to provide proper treatment and further insights about the disorder itself, sufficient qualitative longitudinal measurements from patients are necessary. Therefore, traditional assessment strategies (e.g., clinical interviews or pen-and-paper questionnaires) may be inappropriate to achieve the latter, for example due to the retrospective bias of a patient or the overall cost to conduct such large scale trials [35, 36]. The *TrackYourTinnitus* platform, therefore, follows a mobile crowd sensing approach, which allows to gather large amounts of longitudinal patient data using smart mobile devices. Dedicated mobile applications enable affected patients to record fluctuations of tinnitus symptoms in their everyday lives. Thereby, data collection takes place via dedicated self-assessment questionnaires. In addition to the latter, the *TrackYourTinnitus* mobile application makes use of smartphone-internal sensing capabilities in order to enrich self-reports with contextual information. In detail, the application uses the built-in microphone, to measure the pressure of environmental sound while a patient completes a questionnaire.

Another example where capturing sensor data could be useful, is for monitoring depression symptoms. *Cao et al.* [37] investigated on whether monitoring depression symptoms for people in their adolescence using smartphone applications is useful, compared with other clinical psychometric instruments (e.g., *PHQ-9*). Therefore, over an eight week period, self-reports, sensor data and evaluations from parents were conducted on a daily basis from recruited families with adolescent patients diagnosed with major depressive disorder. Thereby, the mobile application collected a variety of different sensor measurements. The latter included mobility measurements (e.g., step counter, *GPS* coordinates) as well as measurements for social interactions (e.g., *SMS* frequency and call duration). Meanwhile, measurements for baseline depression and anxiety symptoms were taken once every two weeks, using traditional clinical psychometric instruments. The study showed, that by combining the data collected through self-reports with sensor readings from the mobile mobile device, the *PHQ-9* score could be predicted with an accuracy of 88 %. By taking evaluations from the parents of a patient into consideration, the accuracy further increased.

5 Towards a Generic Sensor Framework

Gathering sensor data may be indispensable in mobile data collection scenarios. As elaborated in previous chapters, integrating sensors into the data collection process comes with a number of benefits for both, data collectors (e.g., researchers, clinical staff, etc.) as well as for the people whose data is collected (e.g., study participants, patients, etc.).

Smart mobile devices, used in mobile data collection scenarios, comprise a variety of internal sensing capabilities as well as interfaces to connect with external sensing devices. However, there is no common way of addressing all of these sensors in a generic way. Sensors differ in their type of connection (e.g., internal, wired or wireless), communication protocols, interaction paradigms to collect data and their output format. Depending on the underlying mobile platform (e.g., *iOS* or *Android*), addressing sensors requires the use of different, platform-specific APIs. This imposes massive challenges to application developers as it requires significant knowledge about the sensors themselves as well as the underlying mobile platforms and their APIs to address sensors.

Existing mobile application used for data collection purposes may already access sensors to gather data, nevertheless, they often use dedicated, application-specific implementations to do so. The latter makes it especially hard to further maintain said applications and reuse functionality within other applications. While there exist libraries and frameworks that aim to provide a generic way of addressing a broad spectrum of available sensors (see Chapter 3), the latter may have downsides or lack certain functionality required for specific application scenarios.

To cope with this issue, this thesis is concerned with the development of a sensor framework that may be integrated within existing data collection applications on multiple

platforms in order to generically address sensors in an easy and abstract way. Section 5.1, therefore, specifies requirements such a framework has to fulfill in order to be suitable for a wide range of different application scenarios. With regard to the latter, a general framework architecture, which may act as a reference frame for the actual development of the framework is presented in Section 5.2.

5.1 Requirements

In this section, the most important requirements the mobile sensing framework has to meet are elaborated. Thereby, advantages and drawbacks of existing solutions from Chapter 3 as well as requirements derived from specific application scenarios from Chapter 4, are taken into consideration. Requirements are defined in the following sections and are categorized as either functional or non-functional requirements.

5.1.1 Functional Requirements

The requirements defined in this section describe features and functionality the sensor framework to be implemented has to provide.

FR#1 **Support device internal sensors:**
Modern smart mobile devices are equipped with a rich set of on-board sensing capabilities. Depending on the device, sensors may range from cameras and microphones up until motion, environment and position sensors (e.g., accelerometer, photometer, magnetometer, etc.). The framework should provide ways to access and allow for gathering data from the latter.

FR#2 **Support external sensors and devices:**
Smart mobile devices offer a number of connectivity options to device external resources and are able to communicate with them via different protocols. Thereby, connection may be wired (e.g., *USB*) or wireless (e.g., *Bluetooth* or *WiFi*). The framework should provide developers the opportunity to establish connection with,

and, communicate with external sensing devices connected to a smart mobile device.

FR#3 **Allow for fine-grained sensor-specific configuration at run-time:**
Not only do sensors differ in the type of data they are measuring, but also in how their behavior may be adjusted in order to have more use-case specific sensing outcomes (e.g., adjusting sampling frequency). Hence, sensors within the framework should not have hard-coded, predefined configurations. Rather, configurations should be passed, when requesting data from a specific sensor at run-time. The latter allows for a more versatile use of the framework within different application scenarios.

FR#4 **Allow for registration of sensors at run-time:**
Since there exist tons of different sensors which can be addressed via smart mobile devices, it is impossible to provide dedicated predefined implementations for every single one of them. While providing a set of default sensors for common application scenarios, the framework should allow for registering and communicating with framework-compliant custom sensor implementations created within a host application.

FR#5 **Support different sensor interaction patterns:**
Since sensors may differ in what data they provide and how they are providing it, addressing every sensor the same way might not be the best solution. For example, measuring device acceleration requires frequent and continuous measuring while measuring a devices current location only requires a single request for *GPS* coordinates. To conform with said peculiarities of different sensors, the framework should support multiple ways of interacting with specific sensors. Thereby, sensors within the framework should support at least one particular interaction pattern.

FR#6 **Offline Usage:**
As in many application scenarios a stable internet connection may not be guaranteed [38], the framework itself should follow an 'offline first' approach. Therefore, sensor implementations within the framework, in general, should not require an internet connection for accessing data.

5.1.2 Non Functional Requirements

Requirements defined within this section are not concerned with specific functionality or features the framework should provide, but rather with general characteristics the framework should have.

NFR#1 **Appropriate output format for sensor data:**
Data gathered from sensors should be formatted in a meaningful way. The output data should be both, suitable for further digital processing as well as easy to read and understand for human beings. Following this, data may be processed, analyzed or displayed within the host application. Further, using a format that is easy to read (e.g., *JSON*) instead of raw sensor data formats (e.g., raw byte strings) enables non tech-savvy people to better understand sensor outputs without requiring knowledge about sophisticated sensor specifications.

NFR#2 **Extensibility:**
The framework should be designed in a way that allows for easily extending it with additional functionality. The effort imposed on application developers to integrate custom sensors into the framework should be kept minimal. To achieve the latter, the framework should provide sufficient methods and tooling.

NFR#3 **Fault Tolerance:**
By providing a generic way of accessing sensors on smart mobile devices, certain error scenarios have to be taken into consideration. Some requested sensors may not be available on a specific device or certain permissions required for accessing sensor data are not granted by the user. Also, sensors themselves are prone to errors in numerous ways (e.g., hardware failure). However, possible errors from within the framework should not cause a host application to stop working or even crash. To cope with these issues, there should be ways for the framework to properly communicate errors to the application.

NFR#4 **Framework integration:**
Since the main purpose of the framework is to enhance existing mobile applications with sensing capabilities, integrating and using it within an existing mobile applica-

tion should be as easy as possible. The number of installation and configuration steps required to make the framework work within a mobile application should be kept minimal. In a best-case scenario, the framework should be integrated in a 'plug-and-play' fashion, with no additional configurations steps needed.

NFR#5 **Support different mobile platforms:**
As elaborated in Chapter 3, existing sensor framework approaches are often targeted towards one specific mobile platform. As a result, application developers may neglect a specific framework since it is essential nowadays to provide mobile applications for all major platforms. Hence, the framework developed in the course of this thesis should be implemented in a way, so that it can be integrated into mobile applications running on different platforms (e.g., *iOS*, *Android*, etc.).

5.2 Framework Architecture

With the requirements elaborated in Section 5.1 in mind, a general architecture for the framework to be implemented, was designed. An overview of the latter is presented in Figure 5.1.

To begin with, the sensor framework is designed in a modular way which allows for different parts of the framework to be easily adjusted or extended according to application specific needs. The framework itself should be a module comprising all necessities and functionality to run properly after integration into an existing host application. Further, said framework module should expose interfaces and building blocks, which may be used to perform fine-grained adjustments or enhance framework functionality with custom, application-specific framework extensions (e.g., additional sensor implementations).

Communication between a host application and sensors within the framework should not take place directly, as it may lead to unwanted side effects and development complexity when addressing multiple sensors within different parts of the application. Rather, there should be a central unit (see Figure 5.1, *Sensor Framework Manager*) which provides an interface for a host application to address every sensor available within the framework.

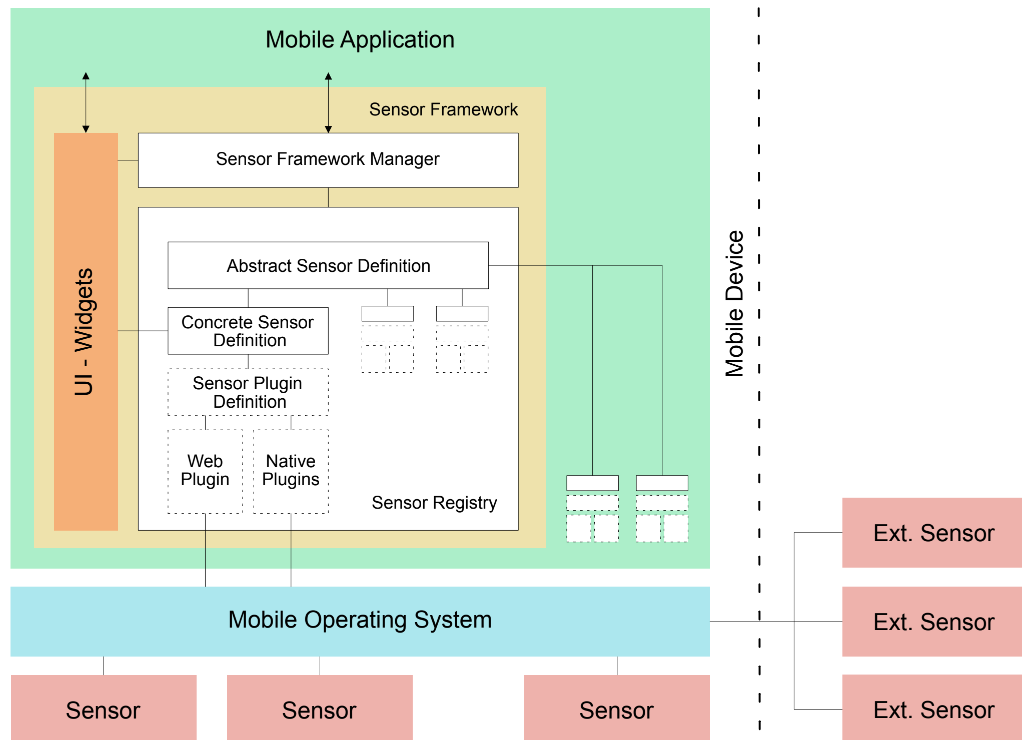


Figure 5.1: Generic Sensor Framework Architecture

In order to address specific sensor implementations in a generic way, proper abstractions from concrete sensor behavior have to be made. Therefore, the framework has to provide a generalized blueprint (see Figure 5.1, *Abstract Sensor Definition*), defining basic interfaces and behavior for sensors within the framework. The latter may also be used when developing application-specific, custom sensor implementations. By doing so, it can be guaranteed that custom implementations can also be addressed through the framework. While the *Abstract Sensor Definition* provides a reference frame for all sensors, abstract functionality may be refined within dedicated, sensor-specific implementations (see Figure 5.1, *Concrete Sensor Definition*). The communication with the actual sensors via platform-specific APIs takes place in dedicated native implementations (see Figure 5.1, *Web Plugin* and *Native Plugins*).

For sensor implementations to be discovered and addressed through the framework, a central unit (see Figure 5.1, *Sensor Registry*) should be in charge of holding references to sensor implementations registered within the framework. The latter may include default implementations from within the framework and application-specific, custom sensor implementations defined outside of the framework.

Finally, the framework may provide a set of user interface components (see Figure 5.1, *UI-Widgets*) that can be embedded directly into the user interface of a host application.

6 Implementation

While previous chapters were more focused on theoretical aspects, this chapter describes the actual implementation of the sensor framework, based on requirements elaborated in Chapter 5. Thereby, Section 6.1 briefly describes the technology stack used to build the framework. Next, relevant aspects concerned with the software architecture of the framework implementation are presented in Section 6.3. Finally, Section 6.4 covers in-depth implementation details of sensors within the framework.

6.1 Technologies

Since *NFR#5* requires the framework to run on various mobile platforms, it was chosen to go with a web technology based implementation approach. The latter allows the framework to run within a regular web browser, but also enables the integration of the framework into mobile applications following a cross-platform development approach, as described in Section 2.2. More specifically, the framework aims to go hand in hand with *Capacitor*-based mobile applications. Therefore, the framework heavily relies on either existing or dedicated, custom *Capacitor* plugins in order to address internal as well as external sensors on various mobile platforms. Since *Capacitor* only provides an application run-time and access to native platform features, application developers are not tied to a specific front-end framework in order to implement the application's business logic and user interface. The choice of a certain framework may also be rejected in favor of using *Vanilla JavaScript*, *HTML* and *CSS*. With regard to the latter, user interface widgets within the framework were built as web components using *Stencil*. As a result, while being tightly coupled to *Capacitor*, the sensor framework is completely independent

when it comes to front-end web frameworks, which further eases possible integration scenarios (*NFR#4*). In detail, the developed sensor framework may be used with the latest state-of-the art front-end web frameworks, such as *Angular* and *React*.

6.2 Capacitor Plugin Implementations

As elaborated before, the developed framework makes use of *Capacitor* plugins in order to address native features on respective platforms. However, since *Capacitor* is still in its early stages, the core plugin set is limited and does not cover the whole spectrum of sensors that may be suitable for specific use case scenarios. To cope with this issue, custom capacitor plugins were implemented whenever the functionality provided by core plugins was insufficient. In detail, plugins for communicating with *Bluetooth Low Energy* peripheral devices (Subsection 6.2.1) and addressing motion, environment and position sensors of a mobile device (Subsection 6.2.2) were developed.

While *Capacitor* core plugins provide native implementations for *Android*, *iOS* and the web by default, the plugins developed in the course of this thesis only include implementations for *Android* and the web. An *iOS* implementation for the custom plugins may surely be possible, but were out of the scope in the context of this thesis.

6.2.1 Bluetooth Low Energy Plugin

Due to its energy efficiency, *Bluetooth Low Energy* enjoys growing popularity as a wireless communication standard for external sensing devices, such as heart rate monitors, pulse oximeters and thermometers, to name a few. In order to be able to connect to the latter through the developed framework (*FR#2*), a custom *Capacitor* plugin, enabling communication with *Bluetooth Low Energy* peripheral devices, was implemented. The peripheral devices themselves may implement one of many application specific *GATT* profiles, defined by the *Bluetooth SIG* [8]. For example, a heart rate monitor may use a corresponding *Heart Rate Profile*. For devices implementing a specific profile, the availability of services and characteristics defined within the respective profile specifica-

tion may be guaranteed. Further, this implies that available services and characteristics comply with their corresponding definitions.

In order to avoid implementing separate plugins to support multiple application specific profiles, the plugin follows a generic implementation approach. Hence, it is compatible with all peripheral devices implementing one of the standard *Bluetooth SIG* application profiles. The plugin itself provides an interface for performing standard *Bluetooth Low Energy* related operations. First of all, it allows for checking whether *Bluetooth Low Energy* is supported by a certain mobile device. As described in Section 2.1, *Bluetooth Low Energy* was introduced as part of the *Bluetooth 4.0* core specification, and thus, may not be available on older devices. Further, surroundings may be scanned for advertising packets from peripheral devices. Thereby, the plugin allows for passing a set of service *UUIDs* in order to limit the scan results to only peripherals which offer services that correspond with the passed *UUIDs*. Scan results contain identifiers of matching peripherals, which may be used to establish connection to or disconnect from a corresponding peripheral device via dedicated `connect()` and `disconnect()` methods provided by the plugin. After establishing a connection the plugin allows for performing standard *GATT* operations on peripheral devices, such as service discovery, reading and writing characteristic values or descriptors as well as enabling and disabling notifications or indications for characteristics supporting the latter.

Within the *Android* implementation of the plugin, *Bluetooth* related features are accessed through the default *Android* `BluetoothManager` service. The latter requires a set of *Bluetooth* as well as location related permissions to be granted by the user in order to work properly. In contrast, the web implementation relies on the *Web Bluetooth API* [39], a specification from the *Web Bluetooth Community Group*, which may be exposed through a web browsers `Navigator` interface. However, the specification is only implemented in a limited set of browsers and is neither a *W3C* standard, nor on the track to become one, at the moment.

The plugin itself, in both, the web and *Android* implementation, returns data, from read operations or as part of notifications/indications, as raw byte values. This raises problems when it comes to further processing and analyzing gathered data. For example,

characteristic values mostly contain multiple pieces of information at once, which are implicitly encoded within one or multiple bytes. Further, the raw byte does not contain information about what it actually refers to. Hence, delivering raw byte data would require sophisticated knowledge about the underlying definition of a characteristic value, for proper processing and analysis. To cope with this issue, the plugin provides a set of transformation methods for selected characteristic values. As depicted in Listing 6.1 for temperature measurements of a thermometer peripheral, those transformation methods receive raw byte data, transform the latter according to the respective characteristic definition, and return processed data as *JSON* objects (*NFR#1*).

```
1  export const TemperatureMeasurementCallback = (data) => {
2      const view = toDataView(data);
3      const flags = view.getUint8(0); // get flags byte at index 0
4      let index = 1;
5      let measurement = {};
6      const unit = (flags & 0x1) ? "F" : "C"; // check if bit at index 0 of flags byte is set
7      const temperature = getFloat32(view, index); // temperature value is encoded in the next 4 bytes
8      measurement = {...measurement, unit, temperature};
9      index += 4;
10     const timestampPresent = flags & 0x2; // get bit at index 1 of flags byte
11
12     if(timestampPresent){
13      // ...
14     }
15     const temperatureTypePresent = flags & 0x4; // get bit at index 2 of flags byte
16     if(temperatureTypePresent){
17         const temperatureType = view.getUint8(index);
18         measurement = {...measurement, temperatureType};
19     }
20     return measurement;
21 };
```

Listing 6.1: Transformation from byte data to *JSON* format for temperature measurements

6.2.2 Internal Sensor Plugin

Most modern smart mobile devices have a variety of built-in sensors. Thereby, the latter may measure acceleration forces along the three axes of a mobile device (e.g., gyroscope, accelerometer), environmental parameters such as the ambient light level or temperature (e.g., photometer, thermometer) or a device's physical position (e.g.,

magnetometer, orientation sensors) [40]. For the purpose of addressing aforementioned sensors within the developed framework, a custom *Capacitor* plugin was implemented. The latter acts as an abstraction layer to platform-specific native implementations, and, therefore, provides a common API to check for availability and activity of a certain sensor as well as for starting and stopping specific sensors, on respective platforms. Further, the plugin may dispatch sensor specific events whenever the value of a sensor changes or its accuracy changes. Those events contain data that may be consumed by the developed framework.

In native *Android* environments, calls to the plugin are dispatched to a dedicated *Java* implementation of the plugin. The latter, in turn, relies on the *Android Sensor Framework* [40] which allows for a generic way of accessing internal sensors of a smart mobile device running *Android*. Thereby, interactions with sensors take place via the `SensorManager` *Android* system service. The service itself provides functionality to get instances of specific system sensors as well as registering/deregistering listeners for the latter. Said listener classes may be defined manually and must implement a common `SensorEventListener` interface, and, therefore, need to provide implementations for `onSensorChanged()` and `onAccuracyChanged()` methods. These methods are called internally by respective sensor implementations whenever its value or accuracy changes. Since the plugin should allow for addressing multiple different sensors, a custom listener class had to be implemented for each sensor. Therefore, common functionality was outsourced to the `AbstractSensorListener` base class, which implements the `SensorEventListener` interface and handles `onSensorChanged()` and `onAccuracyChanged()` functionality. Further, `AbstractSensorListener` defines two abstract methods `getSensorType()` and `toJSON()`. Thereby, `getSensorType()` should return the type of the sensor for which the listener is registered and `toJSON()` should transform sensor readings from `float` arrays into a more readable *JSON* format (complying with *NFR#1*). However, the specific implementation of these methods is outsourced into dedicated sensor listener classes (e.g., `AcceleormeterListener`) which, in turn are derived from the `AbstractSensorListener` base class (see Listing 6.2). The sensor-specific listener implementations can then be instantiated dy-

namically and registered for their corresponding sensor through the `SensorManager` service at run-time.

```
1  public class AccelerometerListener extends AbstractSensorListener {
2
3      public AccelerometerListener(SensingKit kit){
4          super(kit);
5      }
6
7      @Override
8      protected String getSensorType(){
9          return SensorNameResolver.NAME_ACCELEROMETER;
10     }
11
12     @Override
13     protected JSObject toJSON(float[] values){
14         JSObject reading = new JSObject();
15         reading.put(keyX, values[0]);
16         reading.put(keyY, values[1]);
17         reading.put(keyZ, values[2]);
18         return reading;
19     }
20 }
```

Listing 6.2: *Java* implementation of `AccelerometerListener`

When starting an internal sensor through the plugin, a desired sampling frequency (in *Hz*) may be passed. If supported by the respective sensor, this property can be used when registering a listener for the sensor through the `SensorManager` service. Since the *Android Sensor Framework* internally uses time-intervals in microseconds to express the frequency, the property is converted before registration, using Equation 6.1. However it has to be noted, that this parameter only acts as a suggestion and the actual sampling frequency may vary slightly [40].

$$\lfloor \frac{1}{frequency} * 1000 \rfloor \qquad \forall frequency > 0 \tag{6.1}$$

In addition to the *Java* implementation for *Android*, the plugin provides a *JavaScript/-TypeScript* implementation. The latter makes use of the *Generic Sensor API* [41], a *W3C* specification, which is in a *Candidate Reccomendation* state currently and aims to define a framework for exposing sensor data on the web in a consistent way. By now, the *Generic Sensor API* is only implemented within a few modern web browsers and is

not enabled by default. Rather, the API is available as an experimental feature within supporting browsers and has to be enabled manually in the web browser settings by the user. As opposed to the *Android Sensor Framework*, there is no central entity in charge of handling sensor availability and access. Rather, interfaces for specific sensors (e.g., `Accelerometer`, `Magnetometer`, etc.) are exposed within the `window` scope of a web browser directly. Sensors have to be initialized manually by creating an instance of the respective sensor class. Optionally, the sensor classes accept a sampling frequency (in *Hz*) which may be passed as a constructor parameter. After instantiation of a sensor, dedicated handlers can be attached to the instance, in order to define its behavior on activation, in cases of an error occurring or when there is sampling data available for the sensor. For starting and stopping the sensing process for a particular sensor, the sensor class provides respective `start()` and `stop()` methods.

In order to be able to call sensors in a generic way, each sensor type is defined by a unique name, which acts as an identifier across different platforms. For name resolution purposes, each platform specific implementation of the plugin contains a dedicated resolver. The latter assigns platform specific properties to the name for each sensor type. For *Android*, resolver entries include the *Android* specific identification for the sensor type as well as a reference to its corresponding listener class (see Listing 6.3).

```
1  public class SensorNameResolver extends HashMap<String, SensorNameResolverEntry> {
2      public static final String NAME_ACCELEROMETER ="accelerometer";
3          ...
4      public SensorNameResolver(){
5          super();
6          put(NAME_ACCELEROMETER, new SensorNameResolverEntry(){{
7              put(SensorNameResolverEntry.keySensorType, Sensor.TYPE_ACCELEROMETER);
8              put(SensorNameResolverEntry.keyListenerClass, AccelerometerListener.class);
9          }});
10                 ...
11     }
12 }
```

Listing 6.3: `SensorNameResolver` within *Android* Implementation

In contrast, resolver entries within the web implementation of the plugin (see Listing 6.4) include a reference to their corresponding sensor class in the `window` scope as well as a sensor specific `getValue()` method which may be attached to a sensor instance

as a handler in order to extract desired properties whenever the value of the sensor changes. Further, resolver entries contain the maximum sampling frequency allowed for a certain sensor and a list of permissions required for the sensor to work. The latter have to be granted by the user.

```
export const SensorNameResolver = {
    [SensorType.ACCELEROMETER]: {
        class: window.Accelerometer,
        permissions: [SensorPermission.ACCELEROMETER],
        maxFrequency: 60,
        getValue: (sensor: Accelerometer) => {
            const x = sensor.x;
            const y = sensor.y;
            const z = sensor.z;
            return {x,y,z};
        }
    },
        ...
};
```

Listing 6.4: `SensorNameResolver` within Web Implementation

6.3 Software Architecture

The overall software architecture of the developed framework is derived from Section 5.2. This section in particular discusses the three main entities of the framework, highlighted in Figure 6.1. The latter aim to provide an abstract and solid foundation for the entire framework.

At the bottom-most layer, the `Sensor` base class acts as a blueprint for all concrete sensor implementations within the framework. References to the latter are held centrally within the `SensorRegistry`. The `SensorRegistry`, in turn, is used by the `SensorManager` in order to properly resolve and forward requests from a host application to the respective sensor instances.

In the following sections, core concepts and implementation details regarding the `Sensor` base class, `SensorRegistry` and `SensorManager`, are elaborated.

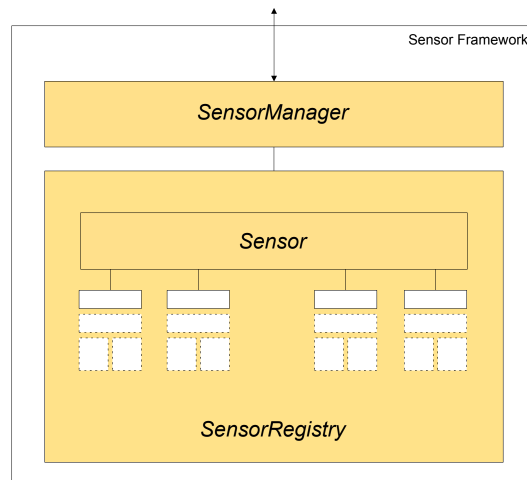


Figure 6.1: Simplified General Software Architecture of the Developed Framework derived from Figure 5.1

6.3.1 Sensor

Within the framework, each physical or software sensor available on a mobile device is represented by a dedicated framework-specific software sensor. The latter reside at the bottom-most layer of the framework and define the communication with the actual sensors in a concrete way. Therefore, they are using platform specific APIs and access patterns which may be further wrapped within *Capacitor* plugins. Said APIs and access patterns may differ greatly between different platforms and sensors. Hence, another abstraction layer is needed to be able to address sensors in a generic way, despite of their characteristic differences. Therefore, a common `Sensor` base class was implemented. For a sensor to work properly within and be addressed through the framework, it has to extend this basic `Sensor` class. The latter defines abstract, sensor related procedures and provides both, interfaces for being addressed through the `SensorManager` as well as hooks and utility methods to be used within concrete sensor implementations. Further, the `Sensor` class implements common functionality related to state management and event handling. Some of the key concepts concerned with the framework specific sensor implementation are described in the course of this section.

Configuration

To be able to properly address a sensor at run-time it must provide a dedicated configuration. The `Sensor` base class, therefore, accepts a `SensorConfig` object as constructor parameter. Said configuration object consists of a `name` as well as an `actions` property. The name property acts as a unique identifier for a certain sensor within the framework. The latter is used to forward calls from an application to a corresponding sensor as well as to provide contextual information within sensor readings. Further, the `actions` property defines a set of actions associated with a certain sensor. Actions, in turn, refer to specific interaction patterns which may be supported by the sensor. Thereby, each action is represented by a `boolean` flag which can be set to `true` if the corresponding sensor supports a certain action. By default, `Sensor` instances do not support any action, so action flags have to be set explicitly.

Sensor Interaction Patterns

When it comes to gathering data from sensors, the framework offers a predefined set of different interaction patterns (*FR#5*, see Figure 6.2). For each possible interaction pattern, the `Sensor` base class provides a dedicated method allowing a host application to initiate interaction with a certain sensor. For enabling concrete sensor implementations (sub-classes of `Sensor`) to adjust their behavior in specific interaction scenarios, the framework follows a hook-based approach. Thereby, if a certain sensor aims to provide functionality for an interaction pattern, a corresponding hook has to be implemented. The latter gets called internally when the interaction is initiated by the host application. Available sensor interaction patterns and their corresponding hooks are briefly described in the following.

get This interaction pattern can be initiated by calling the `get()` method on a sensor instance. Similar to the *Single-Datarequest* pattern described in Section 3.2, on request, the sensor instance is in charge of creating and returning a single measurement (e.g., obtain the current location of a user). Therefore, the underlying sensor class has to im-

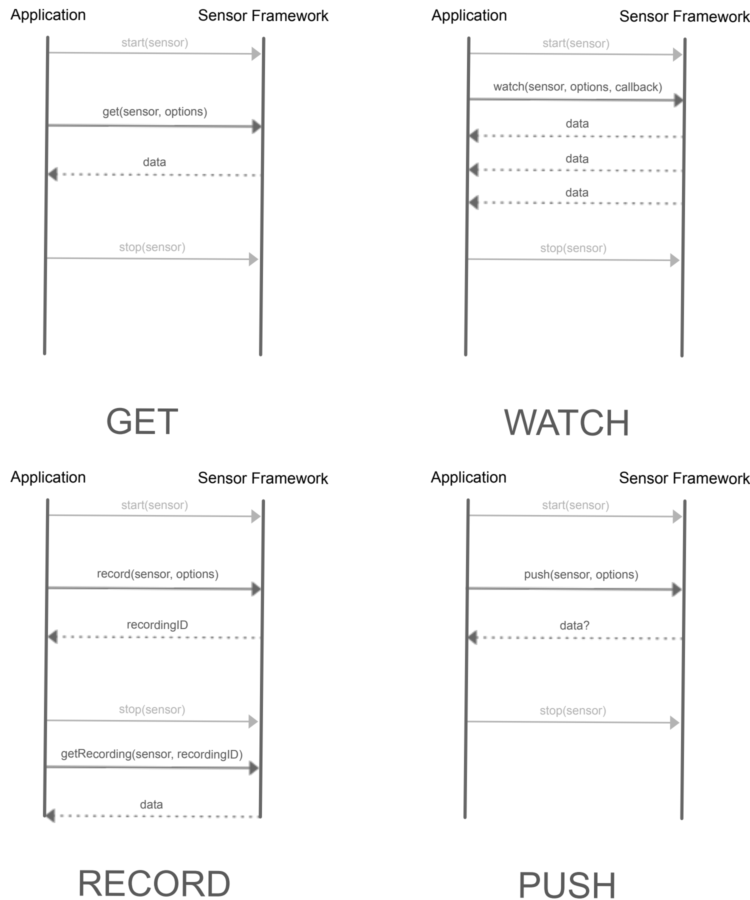


Figure 6.2: Sensor Interaction Patterns

plement the `onGet()` hook, which must return requested measurement as *JSON* object. Further, the `get` flag has to be set within the `actions` property of the corresponding `SensorConfig`.

watch In order to initiate a continuous *stream* of sensor measurements (*Multiple-Dataresponse* in Section 3.2), for example when monitoring heart-rates, one has to call the `watch()` method on a sensor instance. A callback method may be passed, which is triggered whenever the value of a certain sensor changes. Within the underlying class of the sensor, the `onWatch()` hook must be implemented. Within this hook, changes may be propagated upwards by calling the `onSensorDataChanged()` method and

passing the updated value as parameter. Also, the `watch` flag must be part of the `SensorConfig`.

record This pattern shares similarities with the *Recording* pattern from Section 3.2. It allows for gathering sensor data internally until the sensor stops running (e.g., audio or video recordings). The recording can be initiated by calling the `record()` method on a sensor instance supporting the *record* pattern. Therefore, the `onRecord()` hook needs to be implemented within the corresponding sensor class. The hook must return a unique identifier for a given recording. The latter is used later to obtain gathered data after the sensor instance stops running, by passing the identifier to a dedicated `getRecording()` method as a parameter. In order to be able to initiate a recording, the `record` flag has to be set within the `SensorConfig` of the respective sensor class.

push This particular interaction pattern can be seen as a counterpart to the previously described *get* pattern. Rather than requesting a measurement from a certain sensor, data is propagated from the host application to the sensor (e.g., perform *Post* request to sensor via *HTTP*). Thereby, the interaction can be initiated using the `push()` method on a sensor instance. Data to propagate to the sensor may be passed as a parameter. The actual propagation logic has to be implemented within an `onPush()` hook of the given sensor class. Finally, the `push` flag has to be configured accordingly.

All of the aforementioned methods to initiate interaction with sensors additionally accept an optional `options` parameter. These options are sensor specific and may be used to configure the behavior of a sensor or its outputs at run-time (*FR#3*). In order for the `options` object to be available within concrete sensor implementations, it is passed down to the corresponding interaction hook.

In addition to the interaction patterns described above, the `Sensor` base class defines a `start()` and `stop()` method. Corresponding `onStart()` and `onStop()` hooks may be implemented within concrete sensor definitions in order to initially set up a given sensor or perform cleanup tasks on sensor termination. Therefore, after calling `start()` on a sensor instance, the latter should be ready to handle all kinds of supported,

data related interactions. Consequently, after calling `stop()` on a sensor instance, all currently running sensor operations (e.g., *watch* or *record*) should be terminated. Further, the state of the sensor instance should be reset in such a way, that it may be started again.

User-Driven Sensor Configuration

While the framework is designed to run in a headless manner, there may be edge cases, where a manual configuration by a user is required in order for a sensor to run properly. For example, for security purposes, some browser APIs (e.g., `requestDevice()` from the *WebBluetooth API* [39]) require an explicit user interaction to be triggered. To cope with this issue, the framework defines a mechanism, which allows sensors to request a manual configuration from the user via dedicated user interface components (see Figure 6.3). By calling `requestUserConfiguration()` at any given point of time within sub-classes of the `Sensor` base class, a manual configuration can be initiated . This method accepts a `SensorUIConfig` object as parameter. Within the latter, a component as well as properties required for a manual configuration are defined. A framework internal component, namely `SensorConfigurationComponent`, is responsible for displaying and managing user interface widgets for manual configuration tasks. Multiple `SensorUIConfig` objects from different sensor implementations may be registered within the configuration queue of the `SensorConfigurationComponent`. The latter dynamically creates the configuration widgets according to the passed definition. In order to be properly displayed and processed, configuration widgets must follow the *Custom Elements Specification* [42] and must be registered within the browser. Further, configuration widgets must implement the `SensorConfigElement` interface, thus, emit `onSuccess` and `onError` events after successful or unsuccessful configuration. Configuration output data, which may be required for a certain sensor implementation to further operate, is returned to the sensor within the event payload.

Alternatively, to bypass the `SensorConfigurationComponent`, sensor configuration widgets may be directly embedded into the existing user interface. By specifying a `host` element within the `SensorUIConfig`, the display and management of a certain

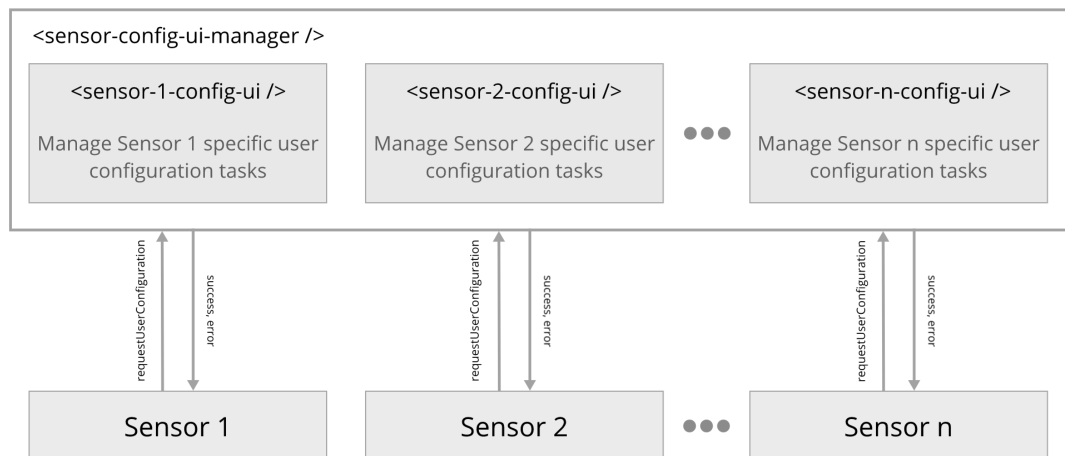


Figure 6.3: Requesting users to manually configure sensors

configuration widget is outsourced to the specified element. The latter must implement the `SensorHostElement` interface and, therefore, provide a `presentUIConfig()` method which is called internally to pass the respective configuration object.

Sensor Data

Due to the versatility and abstract definition of the framework, it allows for the integration of a multitude of different sensors. However, the latter may measure all kinds of different things resulting in highly sensor specific output data. In order to further generalize data gathered through the developed framework, raw output data from interactions with concrete sensor implementations is never sent back to the host application directly. Rather, each output entity is wrapped within a `SampleData` object before it gets forwarded to the host application. Thereby, additional meta information such as the `name` of the corresponding sensor and a timestamp (in milliseconds) is attached to the sensor output. Attached meta information may be valuable when it comes to further processing and analyzing sensor outputs. For example, sensor data may be aggregated or outputs from simultaneously measuring sensors may be analyzed for correlations. Also, sensor data may be used to provide additional context to host application specific measurements.

Error Handling

The framework aims to support a variety of sensors on different mobile platforms. By providing a generalized solution, which abstracts from the underlying platform and its dedicated APIs for addressing sensors, failure scenarios may be predetermined. For example, depending on the mobile device, some sensors or features may not be available despite of an existing framework specific sensor implementation. Further, platform specific ways of addressing sensors and their behaviors can differ greatly, making it hard to cope with errors in a generic way. Also, since the framework allows for the registration of custom, application specific sensor implementations, it is prone to implementation errors caused by third parties and resulting failures at run-time. In order to avoid the host application to stop working due to errors from within the framework (*NFR#3*), the latter follows a defensive implementation approach. In detail, all sensor related exceptions thrown within a concrete sensor implementation (e.g., within dedicated access hooks) are caught at the top-most level in the `Sensor` base class. From there on, the exception is propagated to the host application via a dedicated `onError` event channel. A host application may register dedicated error handlers for sensor instances by calling the `onError()` method and passing the handler as a parameter. The handler is then triggered whenever a failure occurs on the given sensor instance and it is up to the host application to decide how to handle the error. If no error handler is registered for a given sensor, it will just fail silently without causing the host application to crash.

6.3.2 SensorRegistry

In order to properly forward incoming calls from the `SensorManager` to the sensor implementations in charge, the `SensorRegistry` acts as a resolving entity. Therefore, the `SensorRegistry` class maintains an index of specific sensor identifiers and corresponding sensor instances. The identifier, thereby, corresponds with the `name` property within the `SensorConfig` of a given sensor implementation. A sensor instance can be obtained by calling the `getSensor()` method and passing the identifier of the desired sensor implementation as a parameter. Additionally, one may proactively call `isSensorAvailable()` to check whether there already exists a *SensorRegistry* entry

for a given sensor identifier. Further, additional sensor implementations may be registered within the `SensorRegistry` at run-time (*FR#4*) by passing an instance of the corresponding sensor class to the `registerSensor()` method. By doing so, either a new registry entry is created for the passed sensor instance or an existing entry is updated to reference the passed instance. The default `SensorRegistry` instance contains entries for all framework-internal pre-implemented sensors. However, by implementing the `ISensorRegistry` interface, it is possible to create a custom `SensorRegistry` instance. The latter may only reference a subset of the pre-implemented sensors or completely rely on application specific sensor implementations.

6.3.3 SensorManager

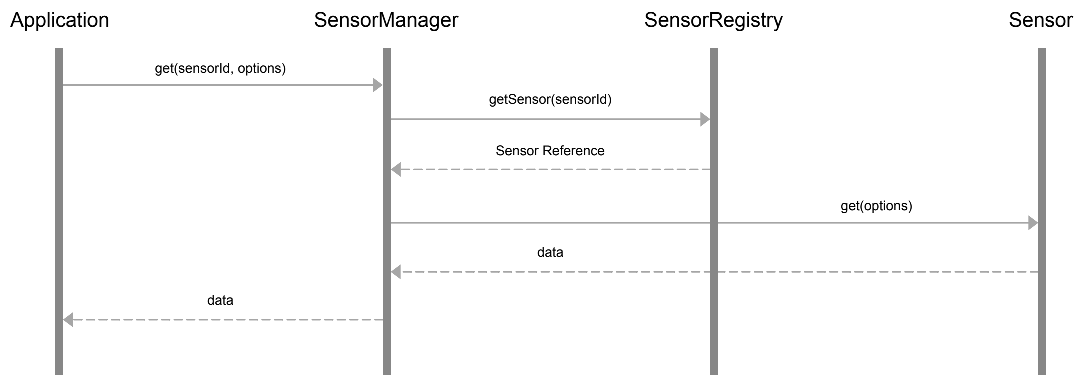


Figure 6.4: Call procedure for a *get* - interaction with all participating entities

The `SensorManager` can be seen as the central entity of the framework. As such it acts as an intermediary between a host application and instances of sensors registered within the framework. Therefore, it provides an interface for the application for starting and stopping a certain sensor, as well as for interacting with the latter using one of the possible interaction patterns defined within the framework. The `SensorManager` itself, thereby, is in charge of forwarding calls from the application to the requested sensors. Illustrative for a *get* - interaction, this procedure is visualized in Figure 6.4. Upon receiving a call from the host application, the `SensorManager` gets a reference

to the corresponding sensor instance from the `SensorRegistry`. For the latter to work properly, calls to the `SensorManager` have to provide a `sensorId` parameter, which is part of the call signature for every sensor related method of the `SensorManager` class. Once the reference to a sensor is successfully obtained, the corresponding action is executed on the sensor instance and the execution result (e.g., sensor data, `SensorListenerHandle`) is returned back to the application. The `SensorManager` is exposed to the application as a singleton instance. This means, that, at any point in time, there exists only one instance of the *SensorManager* throughout the application. Having a central unit for addressing every sensor within the framework may further reduce unwanted side effects from operating on the sensor instances directly within different parts of an application. Finally, in addition to sensor related operations, the `SensorManager` provides an interface for adding sensors to the `SensorRegistry`, thus, allowing a host application to register sensors within the framework at run-time.

6.4 Sensor Implementations

While Section 6.3 described the overall architecture of the developed framework, this section covers details about the concrete implementation of sensors within the framework. In order to eliminate the need for application developers to implement custom sensors, it was aimed to provide a broad spectrum of predefined, ready-to-use sensor implementations. The latter include implementations for addressing smartphone-internal sensors (Subsection 6.4.1) as well as external sensing devices (Subsection 6.4.2). However, despite of having dedicated, framework-specific implementations, some sensors or APIs to address the latter may not be available on every platform or browser environment. To give a brief overview, Table 6.1 summarizes the platform availability for predefined sensor implementations on a wide range of different platforms and web browsers.

6.4.1 Internal Sensors

Smart mobile devices themselves comprise a large number of different sensing utilities suitable for all kinds of application scenarios. In order to address smartphone inter-

Table 6.1: Platform Availability for Sensor Implementations

Sensor	Android	iOS	Chrome	Firefox	Safari	Edge	Opera	Chrome Android	Firefox Android	Opera Android	Safari iOS
Geolocation	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Network Status	✔	✔	✔	✔	✔	✔	✖	✔	✔	?	✔
Microphone	✔	?	✔	✔	?	✔	✔	✔	✔	✔	?
Camera	✔	?	✔	✔	✔	✔	✔	✔	✔	✔	?
Ambient Light	✔	✖	✔	?	?	?	✔	✔	?	✖	?
Gyroscope	✔	⧗	✔	?	?	?	✔	✔	?	✖	?
Magnetometer	✔	⧗	✔	?	?	?	✔	✔	?	✖	?
Accelerometer	✔	⧗	✔	?	?	?	✔	✔	?	✖	?
Linear Acceleration	✔	⧗	✔	?	?	?	✔	✔	?	✖	?
Absolute Orientation	✖	⧗	✔	?	?	?	✔	✔	?	✖	?
Relative Orientation	✖	⧗	✔	?	?	?	✔	✔	?	✖	?
Gravity	✔	⧗	✖	✖	✖	✖	✖	✖	✖	✖	✖
Proximity	✔	⧗	✖	✖	✖	✖	✖	✖	✖	✖	✖
Ambient Pressure	✔	⧗	✖	✖	✖	✖	✖	✖	✖	✖	✖
Ambient Temperature	✔	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
Relative Humidity	✔	✖	✖	✖	✖	✖	✖	✖	✖	✖	✖
Bluetooth Low Energy	✔	⧗	✔	✖	?	?	✔	✔	✖	✔	?
HTTP	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

✔ Available ⧗ Not Implemented ? Availability Unknown ✖ Not Available

nal sensing functionality, the framework provides a wide range of predefined sensor implementations. The latter are presented in the following.

Motion, Position & Environment Sensors

By default, the framework offers support for a wide range of motion, position and environment sensors. The latter internally rely on the custom *Capacitor* plugin implementation described in Subsection 6.2.2. Due to the generic implementation approach of the latter, most of the communication logic between plugin and concrete sensor implementation was outsourced to a common `InternalSensor` class which itself is derived from the `Sensor` base class. Therefore, the concrete implementation of the various sensors available within the framework could be kept quite minimal. They may only contain a respective configuration as well as a dedicated `type` property in order to be properly distinguished by the plugin.

The following sensor implementations exist within the developed framework :

`AbsoluteOrientationSensor`. Measures the physical orientation of a mobile device in relation to the reference coordinate system of the Earth. Measurements returned from the sensor contain *x*, *y*, *z* and *w* values, representing the different components of an orientation quaternion.

`RelativeOrientationSensor`. Measures the physical orientation of a mobile device with no regard to the reference coordinate system of the Earth.

`AccelerometerSensor`. Measures the acceleration forces applied to a mobile device on all three physical axes including the force of gravity. Sensor outputs include *x*, *y* and *z* values in m/s^2.

`LinearAccelerationSensor`. Equivalent to the `AccelerometerSensor` but measurements exclude the force of gravity.

`GravitySensor`. Measures the force of gravity on all physical axes, that is applied to a mobile device. Measurements contain corresponding *x*, *y* and *z* values in m/s^2.

`GyroscopeSensor.` Measures the rate of rotation of a mobile device around all its physical axes. Hence, the outputs from the sensor contain *x*, *y* and *z* values in rad/s.

`ProximitySensor.` Measures the proximity of an object to a mobile device in *cm*. However, some proximity sensors within mobile devices only provide binary values representing near and far.

`AmbientLightSensor.` Measures the ambient light-level within the surroundings of a mobile device and outputs a single illuminance value in *lux*.

`AmbientPressureSensor.` Measures the ambient air pressure around a mobile device. Measurements from the sensor include a single value representing the air pressure in *hPa* or *mbar*.

`AmbientTemperatureSensor.` Measures the ambient room temperature around a mobile device. Sensor outputs include a single value representing the temperature in degrees Celsius.

`MagneticFieldSensor.` Measures the ambient magnetic field for all three physical axes of a mobile device. Measurements from the sensor are composed of corresponding *x*, *y* and *z* values in μT (microtesla).

`RelativeHumiditySensor.` Measures the relative ambient humidity. The resulting measurements contain a single value representing humidity in *%*.

All of the above mentioned sensor implementations only support *watch* interactions. Thereby, additional options may be passed when initiating the interaction. As already described in Subsection 6.2.2, a dedicated sampling frequency in *Hz* may be passed in order to adjust the behavior of a certain sensor. However, if no frequency value is passed, the sensors will fall back to reasonable default values. Further, it has to be mentioned, that some of the sensors are prone to noise interference. As a result, corresponding outputs have to be post-processed in order to eliminate noise. Such noise elimination tasks, however, are out of the scope of the developed framework and may be performed by a host application or when analyzing gathered data at a later point in time.

Geolocation

The `GeolocationSensor` allows for gathering location related data. Therefore, it allows for requesting location data via *get*-action as well as monitoring the location of a mobile device by using the *watch*-action. For accessing said data, the sensor implementation relies on the `Geolocation` *Capacitor* plugin. The latter provides native implementations for *iOS* and *Android* and a dedicated web implementation based on the widespread *Geolocation API* which is exposed through a web browsers `Navigator` interface. The sensor outputs may be adjusted by passing dedicated `GeolocationOptions` to the respective interaction requests. By doing so high accuracy measurements may be enabled, altitude data (if required) may be added to the location measurements or a maximum age for location measurements may be specified. Accordingly, outputs contain a latitude and longitude value along with a corresponding accuracy. If specified and available, the altitude and an accuracy for the latter is part of the measurement. Further, outputs may contain the speed and direction in which a mobile device is moving.

Network Status

A mobile device's network status, for example to find out whether a stable internet connection is granted, may be obtained from the `NetworkStatusSensor`. The latter supports both, *get* and *watch* interactions, and, therefore, allows for requesting the current network status once or continuously watch for network changes. Internally, the sensor implementation relies on the *Network Capacitor* core plugin which offers dedicated native implementations as well as a web based implementation. The latter makes use of the *NavigatorOnLine API* exposed by the web browsers *Navigator* interface. Each measurement gathered from the sensor includes a `connected` property, indicating whether the mobile device is connected to the internet. Further, data may contain the type of connection (e.g., *Wifi* or *4G*).

Media Recording

In order to address the microphone and camera of a smart mobile device, dedicated sensors for audio (`MicrophoneSensor`) and video (`VideoRecorderSensor`) recording were implemented. Both of them rely on the same set of browser APIs in order to gain access and record media in regular browser environments but also within native environments. For accessing the audio or video stream from a mobile device's microphone or camera, the *MediaDevices API*, which is exposed through the *Navigator* interface of a web browser, was used. The returned stream is then captured using the *MediaStream Recording API*. By default, the *MediaDevices API* allows for specifying a set of `MediaStreamConstraints` in order to properly configure the stream. The latter also serve as options for the respective sensor implementations. Thereby, for video recordings it may be specified which camera to use for the stream (e.g., regular or front-facing camera), a set of acceptable or required aspect ratios for the video and a proposed frame rate (in frames per second). For audio streams, in turn, properties such as a proposed sampling rate, sample size and volume may be set. Also, if required and supported, echo cancellation and noise suppression may be activated by specifying respective attributes within the options passed to the sensor instance.

Since the same set of APIs is used for both sensor implementations, an abstract `MediaRecordingSensor` class, defining the communication with respective browser APIs, was implemented. This way the actual implementations of `MicrophoneSensor` and `VideoRecorderSensor` could be kept quite simple. Both sensor implementations support *recording*-interactions and return a `File` object containing recorded audio or video data. The latter may be then uploaded to a server or stored on the mobile device locally. Further, the output contains a browser internal *Object URL* which may be used to directly embed captured media into the user interface of the host application.

6.4.2 External Sensors

As smart mobile devices offer a variety of wireless but also wired connectivity options, external sensing devices may be addressed in numerous ways. In order to achieve

a common behavior to address external sensors, despite of the type of connection they are using for communication, a refined abstraction to the *Sensor* base class, an `ExternalSensor` class, was defined. The latter defines two abstract methods, `connect()` and `disconnect()`, which have to be implemented within respective sensor sub-classes. The `ExternalSensor` class then overrides the `start()` and `stop()` method from the base class by inserting `connect()` and `disconnect()` calls after sensor initialization and before cleanup operations are performed. As a result, external sensors may be treated in the exact same way as internal sensors or other external sensors using different types of connection (e.g., *Bluetooth* and *USB*).

Bluetooth Low Energy Devices

To be able to connect with external sensing devices via *Bluetooth Low Energy*, a dedicated abstract `BleSensor` class, which extends the `ExternalSensor` class, was developed. The `BleSensor` class, thereby, is responsible for the entire communication with peripheral devices, starting from establishing a connection, performing supported *GATT* operations, to releasing the connection again. In order to perform all of the aforementioned operations, the custom *Capacitor* plugin, presented in Subsection 6.2.1, was used. The *BleSensor* class offers support for *get*- , *push*- and *watch*-interactions by default. Thereby, the latter correspond with different *GATT* operations. In detail, the initiation of a *get*-interaction corresponds with a *GATT read* operation, *push*-interactions correspond with *GATT write* operations and *watch*-interactions enable notifications or indications on a certain peripheral. However, before performing these operations, the peripheral has to be discovered and connection to the latter has to be established. This procedure takes place within the `connect()` method. As described before, within web environments, a scan for devices nearby must be triggered through an user interaction. Also, there may be multiple devices nearby, which match the given scan criteria, thus, a decision from the user is required in order to connect to a specific peripheral. Therefore, the `BleSensor` class relies on the framework-internal mechanism for user-driven sensor configuration, presented in Section 6.3.1. A dedicated web component, built with *Stencil*, is in charge of scanning for peripheral devices nearby and presenting a list of discovered

devices to the user. Once the user selects a peripheral from the list, its device identifier (e.g., *MAC-Address*) is returned to the sensor in charge and the latter can connect to the peripheral using the given identifier. Both, the request for user configuration as well as the actual connection establishment take place within the `connect()` method of the `BleSensor` class. Since all the communication logic is implemented within the `BleSensor` class, concrete sensor implementations only have to provide a configuration and define a service as well as a characteristic *UUID*, specifying the type of data to be gathered from a peripheral. While in theory, concrete implementations for every standard *GATT* application profile could be implemented, the framework presented in this thesis, by default, only provides a selected few that may be suitable for data collection scenarios. The latter are described below :

`BleBloodPressureSensor.` This sensor implementation allows for communication with blood pressure monitor devices implementing the standard *Blood Pressure Profile*. Therefore, it supports *watch*-interactions in order to receive *Blood Pressure Measurement* indications from the *Blood Pressure Service*. According to the *Blood Pressure Measurement* characteristic, output values contain the systolic, diastolic and mean arterial pressure, along with a corresponding unit (kPa or $mmHg$). Further, measurements may contain a current pulse rate as well as a status object. The latter may contain details about the measurement itself (e.g., body movement detection, irregular pulse detection or whether or not the measurement took place at an improper position).

`BleHeartRateSensor.` This sensor allows for receiving *Heart Rate Measurement* notifications, from heart rate monitors implementing the standard *Heart Rate Profile*. Therefore, it supports *watch*-interactions. The outputs of the sensor are conform with the respective *Heart Rate Measurement* characteristic definition. They contain the actual heart rate value (in beats per minute). Further, they may contain corresponding *RR*-interval lengths and energy expenditure estimations.

`BleTemperatureSensor.` This sensor allows for addressing health thermometers implementing the standard *Health Thermometer Profile*. In order to subscribe to *Temperature Measurements* from the *Health Thermometer Service*, the sensor

implementation supports *watch*-interactions. Thereby, *Temperature Measurements* contain the temperature value itself as well as a corresponding unit (e.g., Celsius or Fahrenheit). In addition, the type of the temperature, a number representing the body area where the measurement took place (e.g., armpit, ear lobe, rectum, etc.), may be part of the measurement.

`BleWeightScaleSensor`. For weight scales implementing the standard *Weight Scale Profile*, this sensor allows for receiving *Weight Measurement* indications from a corresponding *Weight Scale Service*. Indications from weight scale peripherals may be gathered using *watch*-interactions. Measurements contain the units for weight and height used by the weight scale along with the actual weight value. Further, if stored on the peripheral, measurements may contain the height of a user and a resulting body mass index.

HTTP

The `HttpSensor` allows for communication with sensors over the internet. For example, it may be used to gather temperature data from a web based weather service or to interact with the *Google Fit REST API*, as described in Section 3.3. To achieve the latter it internally uses the *Fetch API*, which allows for performing common *HTTP* requests from within a web browser. The sensor offers support for both, *get*- and *push*-interaction. Thereby, initiating a *get*-interaction corresponds with a *HTTP GET* request whereas *push*-interactions internally perform *HTTP POST* requests. Respective requests may be configured by passing dedicated options when initiating interaction with the sensor. For both interaction schemes, an `uri` property, specifying the endpoint for the *HTTP* request, is required. Further, request specific *HTTP* headers and a strategy for cached data may be specified. For *get*-interactions in particular, additional query parameters, for a more fine grained description of the requested resource, can be provided within the sensor options. In *push* - interaction scenarios, data passed to the `HttpSensor` is embedded in the request body of the corresponding *HTTP POST* request. The sensor outputs are constructed according to the response data from respective *HTTP* requests. Thereby, output data contains the response *URI* and *HTTP* status code of the response.

Further, data from the response body is attached to the sensor output either in a textual or *JSON* format.

7

Enhancing Mobile Applications with Sensing Capabilities

This chapter showcases the usage of the developed framework within, as well as its integration into, existing mobile applications. Therefore, a new hybrid mobile application was created using the *Ionic* framework. The latter uses *Capacitor* as application run-time on mobile devices and *Angular*, a common front-end web framework, for implementing the application's business logic. In the first place, Section 7.1 describes the steps necessary in order to set up the framework within the mobile application. Next, Section 7.2 elaborates on different ways of addressing senors within the application using the framework. Finally, an example of extending the framework with additional, application-specific sensors, is given in Section 7.3.

7.1 Framework Setup within Application

Within the application development environment, the framework may be installed via common *NodeJS* package management tools (e.g., *NPM*, *Yarn*, etc.). Also, the latter are in charge of automatically installing all required framework dependencies within the application.

Since the developed framework uses custom native plugins in order to access internal sensors and *Bluetooth Low Energy* functionality of a mobile device, said plugins must be registered within the application. While the *Capacitor* core plugins are registered within the application by default, custom plugins have to be registered manually. Listing 7.1 shows how custom plugins can be registered within *Android* applications. Thereby, the

common way is to pass references to the respective native class definition of a certain plugin when initializing the application's `MainActivity`.

```
1  public class MainActivity extends BridgeActivity {
2    @Override
3    public void onCreate(Bundle savedInstanceState) {
4      super.onCreate(savedInstanceState);
5
6      this.init(
7      savedInstanceState,
8      new ArrayList<Class<? extends Plugin>>()
9      {{
10       add(BluetoothLEClient.class);
11       add(SensingKit.class);
12     }});
13   }
14 }
```

Listing 7.1: Registering Custom Native Plugins within MainActivity.java

In order to be properly displayed within web browsers or web views, the web components defined within the framework (e.g., UI widgets for sensor configuration) have to be made accessible within the application. For component libraries built with *Stencil*, the `defineCustomElements()` utility method, which allows for automatically registering library internal web components, is generated and exported by default. Ideally, `defineCustomElements()` is called at the top-most level of the application hierarchy (see Listing 7.2), making the web components accessible to the application as soon as it starts running.

```
1 import {defineCustomElements} from 'sensors/loader';
2 (async () => await defineCustomElements(window))();
```

Listing 7.2: Registering *Stencil*-built Web Components within *main.ts*

For *Angular* projects in particular, simply registering the web components from the framework is not sufficient when intending to use them within templates of *Angular* components. When using these web components within *Angular* templates, the containing module has to include the `CUSTOM_ELEMENTS_SCHEMA` in order to not cause compilation errors. However, this last step only relates to *Angular* development and, therefore, may not be necessary when relying on different front-end web frameworks (e.g., *React* or *Vue*).

7.2 Addressing Sensors

While the previous section dealt with setting up the framework within a mobile application, this section is focused on accessing sensor data through the framework. After successfully setting up the framework, there exist two ways of interacting with the framework at run-time. For demonstration purposes, two *Angular* components were exemplarily implemented covering both ways of interacting with two different sensors.

7.2.1 Sensor Access via SensorFrameworkManager

The common approach of accessing sensor data through the sensor framework is by addressing the `SensorFrameworkManager` directly. In order to demonstrate this approach, an *Angular* component for displaying a mobile device's current location, by combining the sensor framework with the *Google Maps JavaScript API*, was implemented (see. Listing A.1). Thereby, the position should be set initially and updated whenever the location of the device changes.

After importing the `SensorFramworkManager` instance, the latter can directly be used to address the devices *GPS* capabilities. Within the `ngAfterViewInit()` *Angular*

lifecycle hook, the corresponding sensor is set up through a `start()` call and the users initial position is gathered by calling `get` method on the `SensorFrameworkManager` instance and passing the name of the sensor as well as sensor specific options. Further, in order to receive updates whenever the location of the user changes, a callback is passed to the `watch()` call, which resets the components `position` property and adjusts the map to display the devices updated location. In order to avoid unnecessary resource consumption, the sensor should be stopped as soon as the component is destroyed. Therefore, within the `ngOnDestroy()` lifecycle, the sensor listener is removed, and the sensor is halted by calling the `stop()` method on the `SensorFrameworkManager` instance.

7.2.2 Sensor Access via Web Component

The way of addressing sensors described in Subsection 7.2.1 requires direct interaction with the `SensorFrameworkManager`. The second approach of accessing sensors through the framework offers an even higher level of abstraction, by making use of the custom `HTMLSensorElement` provided by the framework. For demonstration purposes, this approach was used to connect to and access data from a *Bluetooth Low Energy* heart rate monitor.

```
1 <sensor-element
2   sensor="ble-heart-rate"
3   action="watch"
4   scope="local"
5   (sampleData)="setHeartRate($event)">
6 </sensor-element>
```

Listing 7.3: *Custom* `HTMLSensorElement` *within* `heart-rate.component.html`

As indicated in Listing 7.3, the sensor can be set up by simply integrating the *HTMLSensorElement* within the `HeartRateComponent`'s template. Thereby, configuration can

be initialized by setting element properties as needed. Further, `sampleData` events containing sensor data may be intercepted by binding the event to a corresponding handler (see Listing 7.4) within the business logic of the `HeartRateComponent`.

```
setHeartRate(event) {
    const {data} = event.detail;
    // work with gathered data ...
}
```

Listing 7.4: Event Handler for Heart Rate Measurements within `heart-rate.component.ts`

This template-based approach to accessing sensor data is especially handy when it comes to dynamically gathering data from multiple sensors at once. As displayed in Listing 7.5, multiple `HTMLSensorElements` may be created by looping over an array of corresponding configuration objects. Emitted data can then be aggregated and processed by binding events thrown by the sensors to a common event handler.

```
<sensor-element
 *ngFor="let config of sensorConfigurations"
 [sensor]="config.sensor"
 [action]="config.action"
 [options]="config.options"
 (sampleData)="onDataAvailable($event)"
 (error)="onSensorError($event)">
</sensor-element>
```

Listing 7.5: Dynamic Creation of multiple `HTMLSensorElements`

7.3 Extending the Framework with Custom Sensors

In order to demonstrate how to extend the developed framework with additional, application specific sensors, a custom sensor was implemented within the demo application. The purpose of the sensor was to query and monitor the status of a mobile devices battery. Since the sensor implementation is for demonstration purposes only, a plugin implementation with native features was deliberately abandoned. Rather, the sensor solely relies on the `BatteryManager API`.

To begin with, a new `CustomBatterySensor` class extending the `Sensor` base class was created (see Listing A.2). Within the constructor of the `CustomBatterySensor`, a suitable `SensorConfig` is passed to the constructor of the parent class. The configuration contains the name of the sensor through which it may be addressed later, as well as the actions supported by the sensor. As the sensor should allow for querying as well as monitoring the battery status, the `pull` and `watch` flags were set appropriately. For setting up the sensor, a reference to the `BatteryManager`, which is exposed through the web browsers `Navigator` interface, is stored internally within the sensor's `onStart()` hook. Further, since the `pull` and `watch` flags were set, the corresponding `onPull()` and `onWatch()` hooks had to be implemented. Within the `onPull()` hook, the battery status is queried from the `BatteryManager` and selected properties are returned. In contrast, for continuous monitoring, a callback for `onChargingChange` and `onLevelChange` events is registered on the `BatteryManager` instance. As a result, the callback is triggered whenever either a device's battery level or its charging state changes. By calling the `onSensorDataChanged()` method within the callback, all entities subscribed to battery sensor changes are notified and provided with passed sensor data. To avoid unnecessary resource consumption and side effects, the callbacks on the `BatteryManager` instance are released and the reference to the latter is diminished within the sensor's `onStop()` hook. Finally, an new instance of the `CustomBatterySensor` is created and registered within the framework by calling the `registerSensor()` method on the `SensorFrameworkManager`. Once registered within the framework, the sensor can be used application wide via one of the aforementioned access methods.

7.4 Conclusion

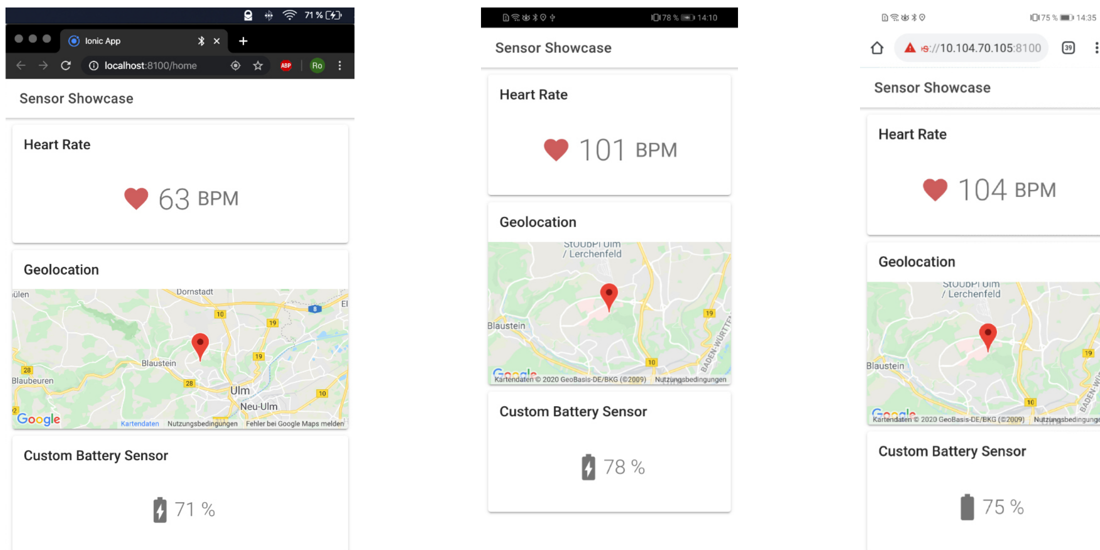


Figure 7.1: Resulting Application running on *Chrome* for *MacOS* (left), as *Android* Application (center) and within *Chrome* Mobile Browser for *Android* (right)

Within this chapter, an in depth description on how to integrate the implemented framework in a mobile application was given. As demanded in *NFR#4*, the setup of the framework within an existing application is rather easy. Nevertheless, a pure 'plug-and-play' solution could not be achieved, as some manual configuration steps are required in order to be able to use the framework properly in an existing application. However, the configuration effort only amounts to adding a few lines of code within the application, which may be reasonable. Regarding *NFR#2*, the effort of extending the framework with an additional custom sensor was quite easy. Since most of the general sensor functionality is outsourced to the `Sensor` base class, only specific operations (e.g., calling respective sensor APIs) have to be implemented within the corresponding hooks provided by the framework.

The resulting application (see Figure 7.1) worked properly on various platforms and devices. For the demonstration of *Bluetooth Low Energy* features, the *nRF Connect Android* application [43], which allows for BLE peripheral simulation on mobile devices, was consulted.

8

Summary

Within this chapter, relevant findings of this thesis are briefly summarized and discussed. Further, an outlook on how the developed sensor framework could be extended with additional features and functionality is given.

In Chapter 2, general aspects regarding the *Bluetooth Low Energy* standard (Section 2.1), cross-platform mobile development strategies (Section 2.2), with special focus on hybrid mobile applications, and the concept of web components (Section 2.3), were introduced. Subsequently, a set of existing mobile sensing frameworks was presented and discussed in Chapter 3. Visited frameworks included *SensingKit* (Section 3.1), an event-based framework approach (Section 3.2) as well as the *Google Fit* framework (Section 3.3). In conclusion, all of the frameworks have a justified existence and may be suitable for a range of use case scenarios. However, under evaluated points of view, most of the frameworks showed slight downsides. While presented frameworks allow for a generic way of addressing different sensors, they may greatly differ in the number of supported sensors, available interaction schemes and extendability options. Most of the frameworks struggled in terms of cross-platform capabilities, meaning having a single framework implementation capable of running within applications on different platforms (e.g., *iOS*, *Android* or web browsers). *SensingKit* and *Google Fit* may aim to support different platforms, with dedicated framework libraries or a *REST API*. Nevertheless, applying the latter can lead to an enormous developing effort and resulting applications may require an active internet connection in order to work properly.
Chapter 4 then elaborated on different use case scenarios, where a mobile sensing framework could find beneficial appliance. While this chapter elaborated on the feasibility of a sensing framework for data collection scenarios in health care and clinical research,

the range of appliance in other domains should not be neglected.
By taking the benefits and drawbacks of existing sensing frameworks into consideration, a set of requirements, the framework to be developed has to fulfill in order to be suitable for a wide range of application scenarios, was elaborated in Chapter 5. According to elaborated requirements, a general framework architecture was set up and described in Section 5.2.
The actual implementation of the framework is described in-depth in the course of Chapter 6. In order to be platform agnostic, the developed framework is entirely based on web technologies, in detail, it is built on top of *Capacitor*. This choice enables the framework to be integrated within mobile applications on different platforms, however, the approach also brings some limitations. As native access to mobile sensor APIs is achieved through dedicated *Capacitor* plugin implementations, the framework is tightly coupled to the *Capacitor* ecosystem. As a result, the framework may only be integrated within regular web applications or mobile applications built on top of the *Capacitor* run-time. Another limitation may arise when it comes to integrating the framework within regular web applications. Since the landscape of internet browsers is far more diverse than the one of mobile operating systems, one can not assume that all browsers in each version support the necessary features to access mobile sensors. Furthermore, many APIs used by the framework to access sensors through a web browser (e.g., *Generic Sensor API* or *Web Bluetooth API*) are marked as experimental features, which have to be enabled manually within the settings of a certain web browser. However, the latter may be a too complex task to perform for regular users. Therefore, in order to exploit the full potential of the developed framework, a more widespread availability of modern web APIs across different browsers is needed. Also, web API implementations within different browsers must become more reliable. For instance, during the development of the framework, a browser update caused the `NetworkStatusSensor` to deliver inaccurate data. Such flaws may be intolerable in production scenarios. However, these issues only relate to the web version of the corresponding sensor implementations, not the native ones.
Finally, to showcase the integration and usage of the developed framework within existing mobile applications, a demo application was implemented in Chapter 7. The framework

setup within the application development environment, as described in Section 7.1, was straight forward and only required a small number of configuration steps. Next, the different ways of addressing sensors through the framework were demonstrated in Section 7.2. Thereby, application developers can choose freely between either accessing sensor data through framework APIs or making use of provided `SensorElement` web component. Section 7.3 then gives insights on how to extend the developed framework with custom sensor implementations. Since most of the general sensor functionality is already provided by the `Sensor` base class, implementing a custom sensor is a quite easy task and requires minimal effort.

8.1 Outlook

As by now, the mobile sensing framework developed in the course of this thesis is still in an early stage. Nevertheless, it already supports a broad range of features and functionality required to fit the needs of many mobile data collection scenarios.
One first step for future development should be the addition of *iOS* implementations for the custom *Capacitor* plugins. While the *Android* and web implementation were sufficient as a proof of concept, having an *iOS* implementation may be crucial in order to fully comply with the sought-after cross-platform approach. Apart from that, potential useful additions to the framework are infinite. For instance, further connectivity options could be implemented within the framework. While at the moment connection to external devices is only possible via *Bluetooth Low Energy* or *HTTP*, one could additionally include *USB* sensors or devices. Therefore, the framework could rely on dedicated platform APIs in native environments and the *WebUSB* API within web browsers. Further, the framework could build bridges to other sensing frameworks, for example by making use of the *Google Fit REST API* to integrate *Google Fit* specific capabilities as described in Chapter 3. Another useful feature could be to allow for gathering sensor data outside of the application run-time. By now, the framework only collects data within an active application (e.g., while a user answers questions within a survey). Though, in some scenarios it might be useful to monitor sensor data while the host application is not running, for example monitoring the accelerometer for fall detection or monitor a patients

heart-rate continuously for irregularities.
However, the maybe most important aspect would be to conduct tests and use the framework within mobile applications in real-world environments. The latter could give valuable insights on the suitability of a cross-platform sensing framework in mobile data collection scenarios as well as how to improve the framework in further iterations.

Bibliography

[1] Seifert, A., Hofer, M., Allemand, M.: Mobile Data Collection: Smart, but Not (Yet) Smart Enough. Frontiers in neuroscience **12** (2018) 971

[2] Schobel, J., Pryss, R., Schlee, W., Probst, T., Gebhardt, D., Schickler, M., Reichert, M.: Development of mobile data collection applications by domain experts: Experimental results from a usability study. In: International Conference on Advanced Information Systems Engineering, Springer (2017) 60–75

[3] Schobel, J., Schickler, M., Pryss, R., Nienhaus, H., Reichert, M.: Using vital sensors in mobile healthcare business applications: challenges, examples, lessons learned. In: International Conference on Web Information Systems and Technologies. (2013) 509–518

[4] Gomez, C., Oller, J., Paradells, J.: Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology. Sensors **12** (2012) 11734–11753

[5] Bhargava, M.: IoT Projects with Bluetooth Low Energy. Packt Publishing Ltd (2017)

[6] Zephyr Project: Bluetooth Stack Architecture. (`https://docs.zephyrproject.org/latest/guides/bluetooth/bluetooth-arch.html`) Accessed: 2020-01-26.

[7] Bluetooth SIG: Bluetooth Core Specification. Bluetooth SIG. (2019) v5.1.

[8] Bluetooth SIG: GATT Specifications. (`https://www.bluetooth.com/specifications/gatt/`) Accessed: 2019-11-17.

[9] Bluetooth SIG: GATT Services. (`https://www.bluetooth.com/specifications/gatt/services/`) Accessed: 2019-11-17.

[10] Bluetooth SIG: GATT Characteristics. (`https://www.bluetooth.com/specifications/gatt/characteristics/`) Accessed: 2019-11-17.

[11] StatCounter: Mobile operating systems' market share worldwide from January 2012 to July 2019. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/ (2019) Accessed: 2019-11-11.

[12] Apache Cordova: Architectural overview of Cordova platform - Apache Cordova. (https://cordova.apache.org/docs/en/latest/guide/overview/index.html#architecture) Accessed: 2019-11-11.

[13] Palmieri, M., Singh, I., Cicchetti, A.: Comparison of cross-platform mobile development tools. In: 2012 16th International Conference on Intelligence in Next Generation Networks, IEEE (2012) 179–186

[14] Drifty Co.: Capacitor: A Cross-platform App Runtime. (https://capacitor.ionicframework.com/docs/)

[15] Drifty Co.: Using Cordova Plugins and Ionic Native. (https://capacitor.ionicframework.com/docs/cordova/using-cordova-plugins) Accessed: 2020-02-07.

[16] developer.mozilla.org: Web Components. (https://developer.mozilla.org/en-US/docs/Web/Web_Components) Accessed: 2020-03-14.

[17] StencilJS: Stencil: A Compiler for Web Components. (https://stenciljs.com/docs/introduction) Accessed: 2019-11-12.

[18] Katevas, K.: SensingKit - A Multi-Platform Mobile Sensing Framework. (https://www.sensingkit.org/) Accessed: 2020-01-19.

[19] Katevas, K.: SensingKit - iOS Reference. (https://www.sensingkit.org/documentation/ios/) Accessed: 2020-01-29.

[20] Katevas, K.: SensingKit - Android Reference. (https://www.sensingkit.org/documentation/android/) Accessed: 2020-01-29.

[21] Jabs, A.: Konzeption eines Event-basierten Sensor-Frameworks zur Datenerhebung auf mobilen Endgeräten (2015) Diploma thesis at Ulm University.

[22] developers.google.com: Google Fit - Platform Overview. (https://developers.google.com/fit/overview) Accessed: 2020-01-13.

[23] developers.google.com: Google Fit - Use Bluetooth Sensors. (https://developers.google.com/fit/android/ble-sensors) Accessed: 2020-01-17.

[24] developers.google.com: Google Fit - Support Additional Sensors. (https://developers.google.com/fit/android/new-sensors) Accessed: 2020-01-19.

[25] developers.google.com: Access Raw Sensor Data. (https://developers.google.com/fit/android/sensors) Accessed: 2020-02-02.

[26] developers.google.com: Work with the Fitness History. (https://developers.google.com/fit/android/history) Accessed: 2020-02-02.

[27] Apple Inc.: HealthKit. (https://developer.apple.com/healthkit/) Accessed: 2020-02-09.

[28] developers.google.com: Google Fit - Custom data types. (https://developers.google.com/fit/datatypes/custom) Accessed: 2020-01-25.

[29] United Nations, Department of Economic and Social Affairs, Population Division (2015): World Population Ageing (2015) (ST/ESA/SER.A/390).

[30] Nguyen, H.H., Mirza, F., Naeem, M.A., Nguyen, M.: A review on IoT healthcare monitoring applications and a vision for transforming sensor data into real-time clinical feedback. In: 2017 IEEE 21st International Conference on Computer Supported Cooperative Work in Design (CSCWD). (2017) 257–262

[31] Bot, B.M., Suver, C., Neto, E.C., Kellen, M., Klein, A., Bare, C., Doerr, M., Pratap, A., Wilbanks, J., Dorsey, E.R., et al.: The mPower study, Parkinson disease mobile data collected using ResearchKit. Scientific data **3** (2016) 1–9

[32] Suh, M.k., Chen, C.A., Woodbridge, J., Tu, M.K., Kim, J.I., Nahapetian, A., Evangelista, L.S., Sarrafzadeh, M.: A remote patient monitoring system for congestive heart failure. Journal of medical systems **35** (2011) 1165–1179

[33] Marko, K.I., Krapf, J.M., Meltzer, A.C., Oh, J., Ganju, N., Martinez, A.G., Sheth, S.G., Gaba, N.D.: Testing the Feasibility of Remote Patient Monitoring in Prenatal Care Using a Mobile App and Connected Devices: A Prospective Observational Trial. JMIR Res Protoc **5** (2016) e200

[34] Bolger, N., Laurenceau, J.P.: Intensive longitudinal methods: An introduction to diary and experience sampling research. Guilford Press (2013)

[35] Pryss, R., Reichert, M., Herrmann, J., Langguth, B., Schlee, W.: Mobile crowd sensing in clinical and psychological trials–a case study. In: 2015 IEEE 28th International Symposium on Computer-Based Medical Systems, IEEE (2015) 23–24

[36] Schlee, W., Pryss, R.C., Probst, T., Schobel, J., Bachmeier, A., Reichert, M., Langguth, B.: Measuring the Moment-to-Moment Variability of Tinnitus: The TrackYourTinnitus Smart Phone App. Frontiers in Aging Neuroscience **8** (2016) 294

[37] Cao, J., Truong, A.L., Banu, S., Shah, A.A., Sabharwal, A., Moukaddam, N.: Tracking and Predicting Depressive Symptoms of Adolescents Using Smartphone-Based Self-Reports, Parental Evaluations, and Passive Phone Sensor Data: Development and Usability Study. JMIR Ment Health **7** (2020) e14045

[38] Schobel, J., Pryss, R., Reichert, M.: Using Smart Mobile Devices for Collecting Structured Data in Clinical Trials: Results From a Large-Scale Case Study. In: 28th IEEE International Symposium on Computer-Based Medical Systems (CBMS 2015), IEEE Computer Society Press (2015) 13–18

[39] Web Bluetooth CG: Web Bluetooth - Draft Community Group Report. https://webbluetoothcg.github.io/web-bluetooth/ (2020) Accessed: 2020-02-24.

[40] developer.android.com: Sensors Overview. (https://developer.android.com/guide/topics/sensors/sensors_overview) Accessed: 2019-12-28.

[41] w3.org: Generic Sensor API. (https://www.w3.org/TR/generic-sensor/) Accessed: 2019-12-30.

[42] WHATWG: Custom elements. (https://html.spec.whatwg.org/multipage/custom-elements.html) Accessed: 2020-03-04.

[43] nordicsemi.com: nRF Connect for Mobile. (https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Connect-for-mobile) Accessed: 2020-06-22.

A

Sources

A.1 Implementation of Geolocation Component

```
1  import ...
2  import {
3  SensorFrameworkManager as SFM,
4  GeolocationData,
5  SensorListenerHandle
6  } from 'sensors';
7
8  @Component({ ... })
9  export class GeolocationComponent {
10
11   private static SENSOR_NAME = 'geolocation';
12   private position: GeolocationData;
13   private listener: SensorListenerHandle;
14   private options = {
15     enableHighAccuracy: true,
16     requireAltitude: true,
17   };
18
19   private map: any;
20
21   private marker: any;
```

```
22
23   @ViewChild('map', {static: true}) mapRef: ElementRef;
24
25   async getGeolocation(): Promise<GeolocationData> {
26     const {data} = await SFM.get(
27         GeolocationComponent.SENSOR_NAME,
28         this.options
29      );
30
31     return data;
32   }
33
34   async watchGeolocation(): Promise<SensorListenerHandle> {
35     const callback = (measurement) => {
36       const {data} = measurement;
37       this.position = data;
38       this.setMarker();
39       this.centerMap();
40     };
41
42     return await SFM.watch(
43         GeolocationComponent.SENSOR_NAME,
44         this.options,
45         callback
46     );
47   }
48
49   private setMarker() {
50     this.marker.setMap(null);
51
52     const position = {
```

```
53        lat: this.position.latitude,
54        lng: this.position.longitude
55      };
56
57      this.marker = new google.maps.Marker({
58          position,
59          map: this.map
60      });
61    }
62
63    private centerMap() {
64      this.map.setCenter(this.marker.getPosition());
65    }
66
67    async ngAfterViewInit() {
68
69      await SFM.start(GeolocationComponent.SENSOR_NAME);
70      this.position = await this.getGeolocation();
71      this.listener = await this.watchGeolocation();
72
73      const position = {
74        lat: this.position.latitude,
75        lng: this.position.longitude
76      };
77
78      this.map = new google.maps.Map(this.mapRef.nativeElement, {
79        center: position,
80        zoom: 12,
81        disableDefaultUI: true,
82      });
83
```

```
84      this.marker = new google.maps.Marker({
85        map: this.map,
86        position,
87      });
88
89    }
90
91    async ngOnDestroy(): Promise<void> {
92      this.listener.remove();
93      await SFM.stop(GeolocationComponent.SENSOR_NAME);
94    }
95  }
```

Listing A.1: `geolocation.component.ts`

A.2 Implementation of Custom Battery Sensor

```
1   import {Sensor, SensorFrameworkManager} from 'sensors';
2
3   class CustomBatterySensor extends Sensor {
4
5       private batteryManager;
6
7       constructor() {
8           super({
9               name: 'battery',
10              actions: {
11                  get: true,
12                  watch: true
13              }
14          });
```

```
15      }
16
17      protected async onStart(): Promise<void> {
18          if (navigator.getBattery !== 'undefined') {
19              this.batteryManager = await navigator.getBattery();
20          } else {
21              throw new Error('Battery API unavailable');
22          }
23      }
24
25      protected async onStop() : Promise<void>{
26          if (this.batteryManager) {
27              this.batteryManager.onlevelchange = undefined;
28              this.batteryManager.onchargingchange = undefined;
29          }
30          this.batteryManager = undefined;
31      }
32
33      protected async onGet(): Promise<CustomBatterySensorData> {
34          const data = {
35              level: this.batteryManager.level,
36              charging: this.batteryManager.charging,
37          };
38          return data;
39      }
40
41      protected async onWatch(): Promise<void> {
42          const handler = () => {
43              const data = {
44                  level: this.batteryManager.level,
45                  charging: this.batteryManager.charging,
```

```
46              };
47              this.onSensorDataChanged(data);
48          };
49          this.batteryManager.onlevelchange = handler;
50          this.batteryManager.onchargingchange = handler;
51      }
52  }
53
54  const Battery = new CustomBatterySensor();
55  SensorFrameworkManager.registerSensor(Battery);
```

Listing A.2: `battery.sensor.ts`

List of Figures

2.1 High-level overview of the *Bluetooth Low Energy* protocol stack [4] 6
2.2 GATT Profile hierarchy [7] 7
2.3 Typical software architecture in hybrid mobile applications [12] 10

3.1 SensingKit Framework Architecture [19, 20] 16
3.2 Architecture of Event-based Sensor Framework [21] 18
3.3 Google Fit high level architecture overview [22] 21

5.1 Generic Sensor Framework Architecture 38

6.1 Simplified General Software Architecture of the Developed Framework derived from Figure 5.1 49
6.2 Sensor Interaction Patterns 51
6.3 Requesting users to manually configure sensors 54
6.4 Call procedure for a *get* - interaction with all participating entities 56

7.1 Resulting Application running on *Chrome* for *MacOS* (left), as *Android* Application (center) and within *Chrome* Mobile Browser for *Android* (right) 73

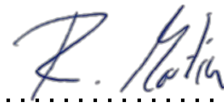
List of Tables

6.1 Platform Availability for Sensor Implementations 58

Name: Robin Martin Matriculation number: 857754

Honesty disclaimer

I hereby affirm that I wrote this thesis independently and that I did not use any other sources or tools than the ones specified.



Ulm, 24.06.2020 ....

Robin Martin