Somnus

Schlaf-Tracking per App

Projekt-Dokumentation

Sarah-Lee Mendenhall
Matthias Klassen
Nele Herzog
Stephanie Scheibe

Mobile Apps for Public Health
Prof. Dr.-Ing. Dabrowski
Angewandte Informatik
Hochschule für Technik und Wirtschaft Berlin

Inhaltsverzeichnis

1 Einleitung

Die App Somnus ermöglicht dem Benutzer, anhand eines Activitytrackers Informationen über seinen Schlaf zu erhalten. In dieser Dokumentation wird der Aufbau und die Funktion der prototypischen Umsetzung beschrieben.

1.1 Überblick Gesamtnetzwerk

Die Anwendung unterteilt sich in verschiedene Architekturen. Die eigentliche Logik, welche die Analyse der gesammelten Aktivitätsdaten vornimmt, ist derzeit in ein externes Backend ausgelagert. Die Interaktion mit dem Anwender und die Visualisierung der Ergebnisse erfolgt in der Android-App. Damit die Analyse der Daten stattfinden kann, werden die folgenden Schritte durchlaufen:

Datenerhebung Das Miband wird über Bluetooth mit dem Smartphone und der Somnus-App verbunden. Es sammelt Daten zur Aktivität anhand eines Accelerometers. Diese Daten werden ständig über die Bluetoothverbindung an die Somnusapp gesendet.

Datenspeicherung Die empfangenen Daten vom Miband werden in eine lokale Datenbank auf dem Smartphone geschrieben.

Datenübermittlung Die gespeicherten Accelerometerdaten werden gesammelt aus der Datenbank extrahiert und als gesammelte CSV Datei einmal täglich zur Auswertung an das externe Backend gesendet.

Datenverarbeitung Das Backend empfängt die CSV-Datei und wertet die empfangenen Daten aus, indem eine Klassifizierung nach Schlaf und Wach stattfindet. Dieses Ergebnis wird als CSV an die App zurückgegeben.

Datendarstellung In der Somnus App werden die Daten aus der CSV in die Datenbank gespeichert. Auf den Analysescreens der App können die Auswertungen grafisch angesehen, verändert oder exportiert werden.

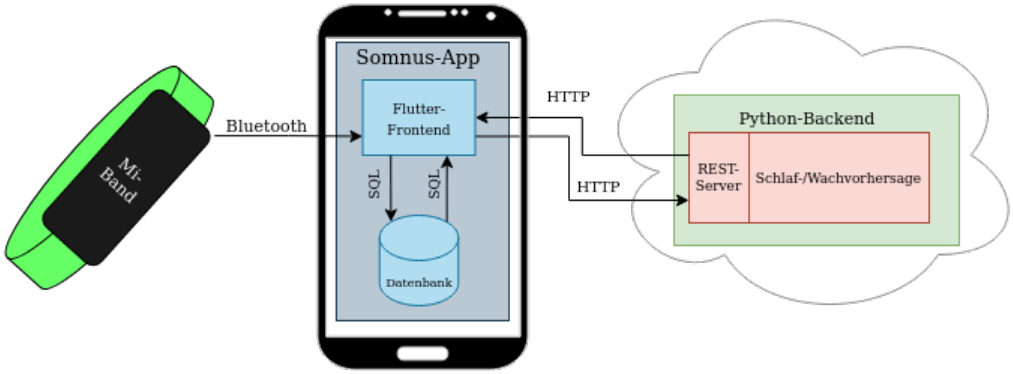

Abbildung 1: Kommunikation Gesamtnetzwerk

1.2 Corporate Design

Für das Corporate Design der App wurden feste Farben definiert, zu sehen in Abbildung 2 und als grundsätzliche Schriftart wird Roboto verwendet.

Abbildung 2: Farbpalette des Corporate Design

2 Flutter

2.1 Setup

Um die App bearbeiten und ausführen zu können, sind die folgenden Schritte erforderlich:

1. Installation notwendiger Tools:

   - Flutter
   - Android Studio
   - Code-Editor, z.B. Intellj, Visual Studio Code oder Emacs

2. Android Setup
   Einrichtung
   Starte Android Studio und folge den Anweisungen des Einrichtungsassistenten. Dadurch werden die neuesten Android SDK-, Android SDK-Befehlszeilentools und Android SDK-Build-Tools installiert, die Flutter bei der Entwicklung für Android benötigt.

   Einrichten eines (physischen) Android-Gerätes oder eines Android Emulators
   Um die App auszuführen zu können, musst du zunächst ein entsprechendes Gerät einrichten und starten. Informationen zum Vorgehen findest du hier.

3. Download der Somnus App von GitLab
   Anschließend kannst du dir den Source Code mittels
   `git clone https://gitlab.com/MatzeK105/somnus.git` oder
   `git clone git@gitlab.com:MatzeK105/somnus.git` aus dem Repository herunterladen.

4. Starten der App
   Gehe in das Verzeichnis `somnus/frontend_somnus/` und öffne hier deinen Editor. Öffne ein Terminal und führe den Befehl `flutter run` aus, um die App zu installieren und auf dem eingerichteten Android-Gerät zu starten. Anschließend kannst du Änderungen im Code vornehmen und diese unmittelbar auf dem Gerät sehen.

2.2 Ordnerstruktur

Verwendete Screens und Widgets befinden sich im Ordner `somnus/frontend/lib/` (siehe Abbildung 3).

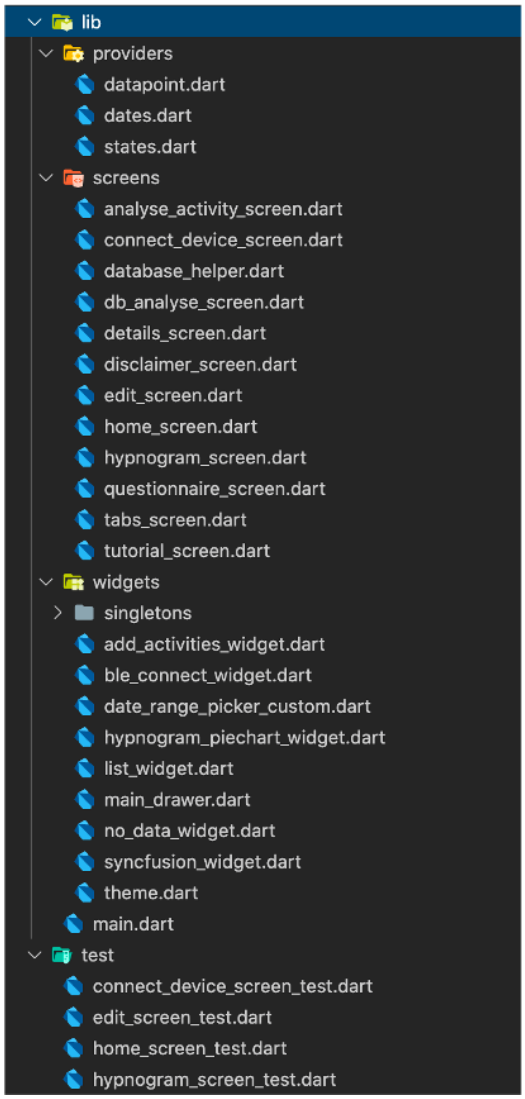

Abbildung 3: Ordnerstruktur der implementierten Screens und Widgets

2.3 Screens und Widgets

Die App setzt sich aus verschiedenen Screens zusammen, in welche mitunter eigene Widgets eingebunden sind. Die Abbildung 4 zeigt die implementierten Screens und Widgets sowie ihren Zusammenhang.

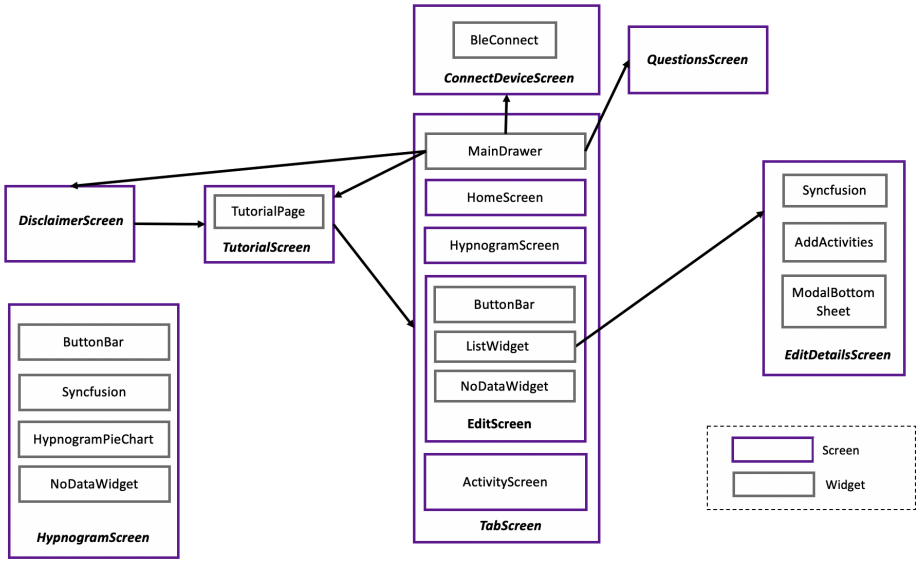

Abbildung 4: Implementierte Screens und Widgets

Im Folgenden werden die Elemente der App kurz vorgestellt.

2.3.1 DisclaimerScreen

Der `DisclaimerScreen` wird beim ersten Öffnen der App einmalig angezeigt und enthält, wie der Name sagt, einen Disclaimer, welcher ausführlich darüber informiert, dass es sich hier nicht um ein Medizinprodukt handelt. Als Vorlage wird der Disclaimer von NALA verwendet, einer App, welche sich an Menschen richtet, die von Neurodermitis betroffen sind.

2.3.2 TutorialScreen

Nach dem Bestätigen des Verständnisses des Disclaimers gelangt man auf den `TutorialScreen`(siehe Abbildung 5), welcher einen Überblick über die verschiedenen Funktionen der Somnus App bietet. Wie der `DisclaimerScreen` wird dieser beim ersten Öffnen der App einmalig angezeigt. Für die Implementierung wurde das Flutter Package `introduction_screen` verwendet und entsprechend angepasst.

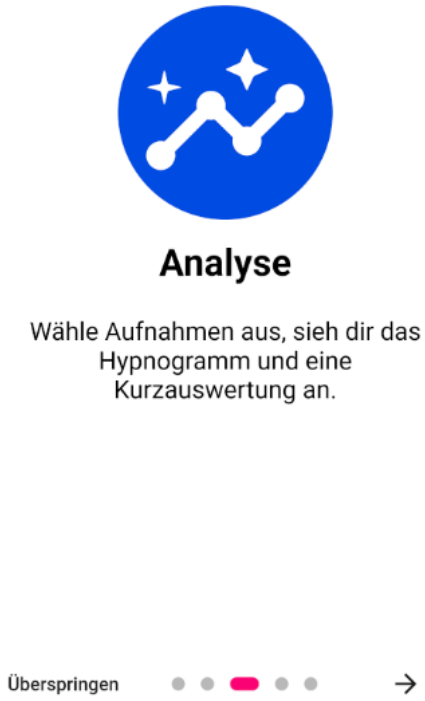

Abbildung 5: Eine Seite des `TutorialScreens`

2.3.3 TabsScreen

Der `TabsScreen` bildet die Grundlage für die App. Ihn ihn sind die Screens eingebettet, zu denen über Registrierkarten (Tabs) navigiert werden kann. Dazu wird eine Liste `_pages1` erstellt, welche die Screens enthält, die in den `TabsScreen` eingebunden werden sollen. Diese wird dann einem `PageView` Widget übergeben, ein `PageController` steuert, welcher Screen sichtbar ist. Das Argument `initialPage` des `PageControllers` bestimmt, welcher Screen beim ersten Erstellen des `PageView` Widgets angezeigt wird. Hier wird als `_selectedPageIndex=0` übegeben, was dafür sorgt, dass dies der `HomeScreen` als erstes Element von `_pages1` ist (siehe Listing 1).

Listing 1: Initialisieren der Liste der in den `TabsScreen` eingebundenen Screens

```
@override
 void initState() {
   super.initState();
   _selectedPageIndex = 0;
   _pages1 = [
     HomeScreen(),
     HypnogramScreen(),
     EditScreen(),
     ActivityScreen(),
   ];
   _pageController = PageController(initialPage:
       _selectedPageIndex);
 }
```

Wie aus dem Code-Beispiel ersichtlich, sind in den `TabsScreen` vier Screens eingebunden, welche im Folgenden kurz vorgestellt werden.

2.3.4 HomeScreen

Abbildung 6: `HomeScreen`

Auf dem `HomeScreen` (siehe Abbildung 6) erhält der Nutzer Informationen über das Datum der letzten Aufzeichnung. Darüber hinaus kann er hier die Verbindung mit dem Mi-Band herstellen oder trennen. Außerdem hat er die Möglichkeit, Aktivitäten und Medikamente hinzuzufügen, welche einen Einfluss auf den Schlaf haben könnten. Diese werden im `ActivityScreen` (siehe Abschnitt 2.3.8) in Bezug auf ihre Wirkung ausgewertet.

Um die einzelenen Kacheln des `HomeScreen`s zu generieren, wurde das Flutter Package `flutter_staggered_grid_view` verwendet, weiches es auf intuitive Weise ermöglicht, `Row`s und `Column`s verschiedener Größe zu implementieren und anzuordnen.

2.3.5 HypnogramScreen

Im `HypnogramScreen` kann der Nutzer sich seine Schlafdaten in Form von Hypnogrammen, wie sie in der Schlafmedizin üblich sind, ansehen. In den `HypnogramScreen` sind vier Widgets eingebunden (siehe Abbildung 7).

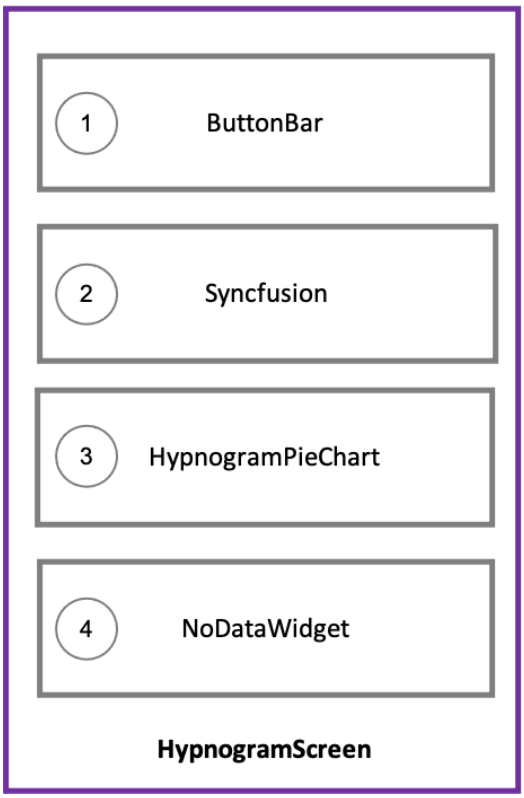

Abbildung 7: `HypnogramScreen` und eingebundene Widgets

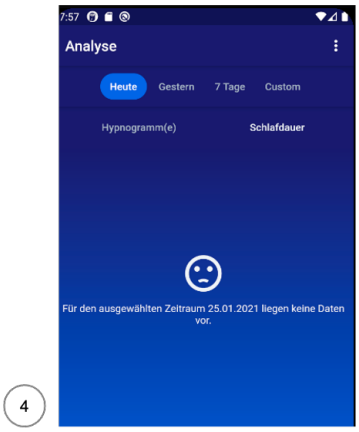

Abbildung 8: `NoDataWidget`

Die Anzeige der Hypnogramme kann nach vordefinierten oder benutzerdefinierten Zeiträumen gefiltert werden. Hat der Nutzer einen Zeitraum ausgewählt, kann er zwischen der Anzeige der Hypnogramm-Daten und einer Auswertung, welche die gesamte Schlaf- beziehungsweise Wachdauer des ausgewählten Zeitraums anzeigt, ausgedrückt entweder in Zeit in Minuten oder prozentual im Verhältnis zur Gesamtdauer der Aufzeichnung des ausgewählten Zeitraums.

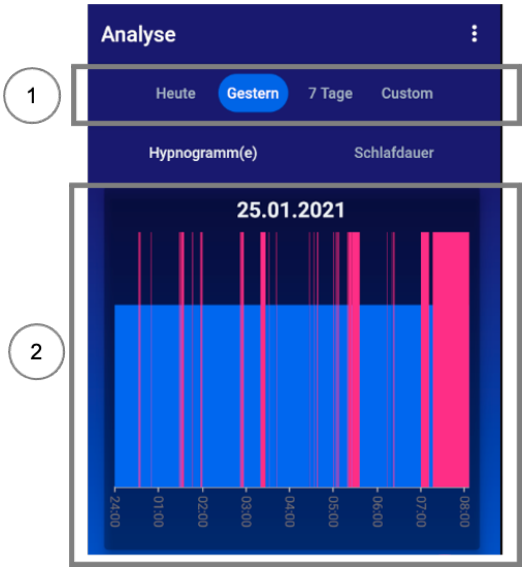

Abbildung 9: `HypnogramScreen`: `ButtonBar` und `Syncfusion` Widget

Abbildung 10: `DateRangePicker`

Liegen für einen ausgewählten Zeitraum keine Daten vor, wird anstelle der Schlafdaten das `NoDateWidget` gerendert, um dem Nutzer ein entprechendes Fedback zu geben (siehe Abbildung 8).

Die Logik zum Filtern und Anzeigen der Daten ist im Widget `ButtonBar` implementiert (siehe Abbildung 9).

Hier werden vier `FlatButtons` gerendert, deren Betätigung die Schlafdaten nach dem ausgewählen Zeitraum selektiert. Neben dem Anzeigen der Daten für den aktuellen und den gestrigen Tag kann ein 7-Tage-Intervall gewählt werden. Benutzerdefinierte Zeiträume können durch Betätigen des Buttons `Custom` gewählt werden. Um hierfür ein Kalender Widget zu implementieren, wird das Futter Package date_range_picker verwendet und angepasst (siehe Abbildung 10).

Das Listing 2 zeigt stellvertretend für den '7 Tage'-Button, wie die Filter-Logik aussieht.

Listing 2: `onPressed`-Funktion des `FlatButtons` '7 Tage' in der `ButtonBar`

```
1                onPressed: () async {
2                  setState(() {
3                    isLoading = true;
4                    _pressedButton3 = true;
5                    _pressedButton1 = false;
6                    _pressedButton2 = false;
7                    _pressedButton4 = false;
8                  });
9                  dataPoints = await getDataSevenDays();
10                 dates = await Provider.of<DataStates>(context,
                       listen: false)
11                     .getEditDataForDateRange(DateTime.now(),
12                         (new DateTime.now()).add(new Duration(days:
                               -7)));
13                 await buildList(dates);
14                 setState(() {
15                   title = formatter.format(
16                           (DateTime.now()).add(new Duration(days:
                                 -7))) +
17                       '␣bis␣' +
18                       formatter.format(DateTime.now()).toString();
19                   sleepData = dataPoints;
20                   syncList = list;
21                   isLoading = false;
22                 });
23               },
```

Wird der Button betätigt, werden unter anderem die Daten der letzten sieben Tages aus der Datenbank abgefragt und in der Variable `dataPoints` gespeichert (Zeile 20). Die Funktion gibt eine Liste vom Typ `<DataPoint>` zurück. Außerdem werden mittels der Funktion `getEditDataForDateRange` alle in der Datenbank enthaltenen verschiedenen Daten für den ausgewählten Zeitraum abgerufen (z.B. 15.01.2021, 16.01.2021, 19.01.2021) und in der Variable `dates` gespeichert. Die Liste `dates` wird dann der Funktion `buildList` übergeben (siehe Listing 3). Diese iteriert über die Liste `dates` und ruft für jedes in der Liste enthaltene Datum die Funktion `getDataForSingleDate` auf. Die Liste an Schlafdaten für ein Datum wird dann im `Syncfusion` Widget verwendet, welches der Liste `list` für jeden in `dates` enthaltenen Eintrag hinzugefügt wird.

Listing 3: Die Funktion `buildList`

```
1            Future<List<Widget>> buildList(dates) async {
2      list = [];
3      for (int i = 0; i < dates.length; i++) {
4        var data = await Provider.of<DataStates>(context, listen:
             false)
5            .getDataForSingleDate(dates[i].date);
6        list.add(
7            Syncfusion(title: formatter.format(dates[i].date),
                 sleepData: data));
8      }
9      return list;
10   }
```

Die zurückgegebene `list` wird dann der der Variablen `syncList` zugewiesen. Diese wird im `HypnogramScreen` in einem `Column` Widget gerendert. Je länger die Liste ist, desto mehr `Syncfusion` Widgets werden also dem Nutzer angezeigt.

Das Widget `Syncfusion` (siehe Abbildung 9) ist zuständig für die Darstellung von einzelnen Hypnogrammen. Es erhält seinen Namen von der Flutter Library syncfusion_flutter_charts, welche für die Darstellung der Daten als Hypnogramm verwendet wird. Hierbei handelt es sich um eine Bibliothek zur Datenvisualisierung, die es ermöglicht, ansprechende, animierte und leistungsfähige Charts zu erstellen. Die Entscheidung für die Verwendung des Packages fiel nicht zuletzt aufgrund der vielen Möglichkeiten der Anpassbarkeit, die es bietet. Für die Darstellung eines Hypnogramms wird das von der Bibliothek zur Verfügung gestellte Widget `SfCartesianChart` verwendet (siehe Listing 4). Bei `Cartesian` Charts handelt es sich um Diagramme mit einer horizontalen und einer vertikalen Achse.

Listing 4: `SfCartesianChart` Widget im `Syncfusion` Widget

```
1
2                    SfCartesianChart(
3                      plotAreaBorderColor: Colors.transparent,
4                      zoomPanBehavior: ZoomPanBehavior(
5                          enablePinching: true,
6                          enableDoubleTapZooming: true,
7                          enableSelectionZooming: true,
8                          selectionRectBorderColor: Colors.red,
9                          selectionRectBorderWidth: 1,
10                         selectionRectColor: Colors.grey),
11                     enableAxisAnimation: true,
12                     tooltipBehavior: TooltipBehavior(enable: true),
```

```
13                    primaryXAxis: DateTimeAxis(
14                      isVisible: true,
15                      majorGridLines: MajorGridLines(width: 0),
16                      dateFormat: formatter,
17                      interval: 1,
18                      labelRotation: 90,
19                      plotBands: <PlotBand>[
20                        /*  Plot band: different height for sleep
                              and awake */
21                        PlotBand(
22                          isVisible: true,
23                          associatedAxisStart: 0.5,
24                          associatedAxisEnd: 0,
25                          shouldRenderAboveSeries: false,
26                          color: colorAsleep,
27                          opacity: 1.0,
28                        ),
29                      ],
30                    ),
31                    primaryYAxis: NumericAxis(
32                      interval: 1,
33                      maximum: 0.7,
34                      isVisible: false,
35                    ),
36                    series: <ChartSeries>[
37                      // Initialize line series
38                      StepAreaSeries<DataPoint, DateTime>(
39                        color: colorAwake,
40                        //Color of awake periods
41                        opacity: 1.0,
42                        dataSource: sleepData,
43                        xValueMapper: (DataPoint sleeps, _) =>
                            sleeps.date,
44                        yValueMapper: (DataPoint sleeps, _) =>
                            sleeps.state,
45                      )
46                    ],
47                  ),
```

Die `SfCartesianChart`-Klasse stellt wiederum eine Reihe von Widgets zur Verfügung, mit denen Diagramme des Typs `Cartesian` erstellt werden können. Hier wird das Widget `StepAreaSeries` verwendet, mit welchem sich Daten gut im Zeitverlauf darstellen lassen (Zeile 38). Auf der x-Achse des Diagrams werden die Zeitangaben im 24-Stunden-Format dargstellt, die y-

Abbildung 11: Als PDF exportiertes Hypnogramm

Achse ist den Zuständen 'Schlaf' oder 'Wach' vorbehalten, die numerisch als 0 beziehungsweise. 1 auf der Achse dargestellt werden.

Es gibt außerdem die Möglichkeit, die ausgewählten Hypnogramme als PDF-Datei zu exportieren und auf dem Gerät zu speichern (siehe Abbildung 11). Hierfür steht in der `AppBar` des `HypnogramScreens` ein `floatingActionButton` zur Verfügung.

Schließlich kann sich der Nutzer für die im gewählten Zeitraum liegenden Aufzeichnungen auch eine Auswertung ansehen, welcher er die Gesamtzeit, die er laut Erkennung schlafend beziehungsweise wach verbracht hat, entnehmen kann. Hierfür wurde das Widget `HypnogramPieChart` implementiert (siehe Abbildung 12).

Hier kann er zwischen der Darstellung in Minuten und prozentualen Anteilen wählen. Für die Umsetzung wird das Flutter Package `pie_chart` genutzt, welches in besonders hohem Maße Möglichkeiten zur Animation bereitstellt.

2.3.6 EditScreen und EditDetailsScreen

Der in den `TabsScreen` eingebundene `EditScreen` ermöglicht es dem Nutzer, Daten nachträglich zu bearbeiten (siehe Abbildung 13).

Dazu wird in den `EditScreen` neben den bereits erwähnten Widgets `ButtonBar` und `NoDataWidget` die Komponente `ListWidget` eingebunden. In dieser wird für jedes Datum im gewählten Zeitraum, für welches Daten in der Datenbank vorliegen, ein `InkWell` Widget gerendert , welches den

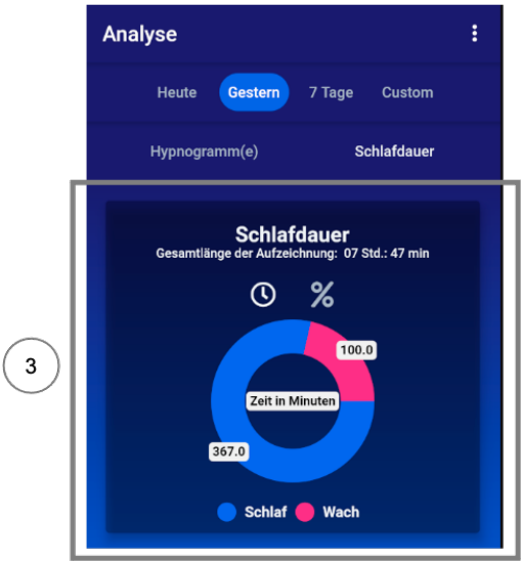

Abbildung 12: `HypnogramScreen`: `HypnogramPieChart` Widget

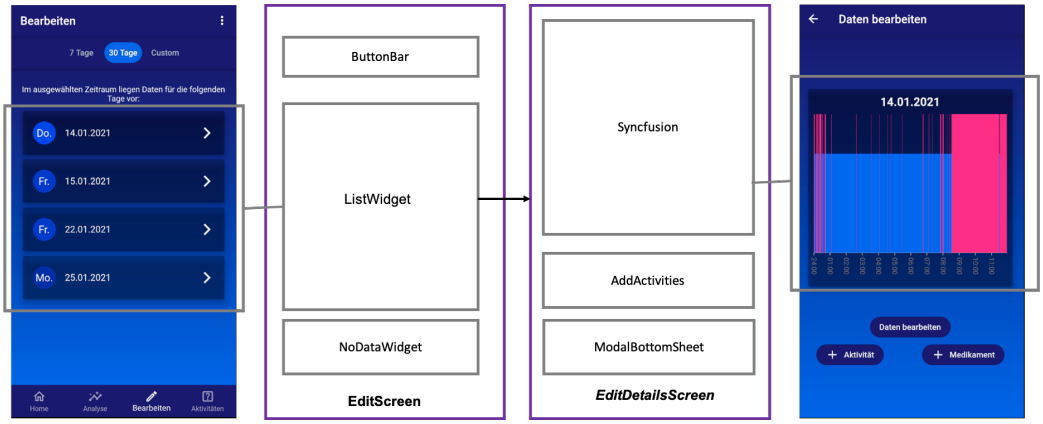

Abbildung 13: `EditScreen` und `EditDetailsScreen`

Nutzer bei Anklicken auf den `EditDetailsScreen` führt. Die Abfrage der Daten, welche im `ListWidget` angezeigt werden, erfolgt über die Funktion `getEditDataForDateRange` (siehe Listing 5), welche wiederum mittels der Funktion `queryDatesDayRange` eine SQL-Anfrage an die Datenbank sendet und das Ergebnis in Form einer Liste zurückgibt.

Listing 5: Funktion `getEditDataForDateRange`

```
1  Future<List<DateEntry>> getEditDataForDateRange(date1, date2)
       async {
2    final DateFormat serverFormater = DateFormat('yyyy-MM-dd');
```

```
    dataDatesFromDB = [];
    final allRows = await dbHelper.queryDatesDayRange(
        serverFormater.format(date2), serverFormater.format(date1));
    allRows.forEach((row) {
      var parsedDate = DateTime.parse(row['date']);
      dataDatesFromDB.add(
        DateEntry(
          date: DateTime(
            parsedDate.year,
            parsedDate.month,
            parsedDate.day,
          ),
        ),
      );
    });
    return dataDatesFromDB;
  }
```

Im `EditDetailsScreen` können die Schlafdaten für einzelne Tage bearbeitet werden. Bei Betätigen des `FlatButtons` `Daten bearbeiten` öffnet sich ein `ModalBottomSheet` Widget, in welchem für eine ausgewählte Zeitspanne die Datenzustände bearbeitet werden können (siehe Abbildung 14).

Außer den Daten selbst können zusätzlich ausgeübte Aktivitäten und eingenommene Medikamente hinzugefügt werden, welche den Schlaf beeinflussen können (siehe Abbildung 15)

Hierzu ist zu sagen, dass die hinzugefügten Aktivitäten und Medikamente aktuell nicht in der Datenbank gespeichert werden. Die in der App sichtbare Funktionalität ist *hardgecoded*.

2.3.7 ConnectDeviceScreen

Um das Fitness-Armband mit dem Smartphone zu koppeln, muss der Nutzer in den `ConnectDeviceScreen` navigieren. Hier werden verfügbare Bluetooth-Low-Energy-Geräte aufgelistet. Der Nutzer soll mit einem Klick auf eines der aufgelisteten Geräte ein Gerät auswählen, zu dem das Smartphone eine Verbindung herstellen soll. Die App prüft nach einer erfolgreichen Verbindung, ob das gekoppelte Gerät ein MiBand 2 ist. Im Falle eines MiBand 2, wird der Authentifizierungsprozess durchgeführt und das Gerät ist gekoppelt, andernfalls wird die Verbindung wieder beendet und der Nutzer darüber informiert, dass das Gerät inkompatibel ist.

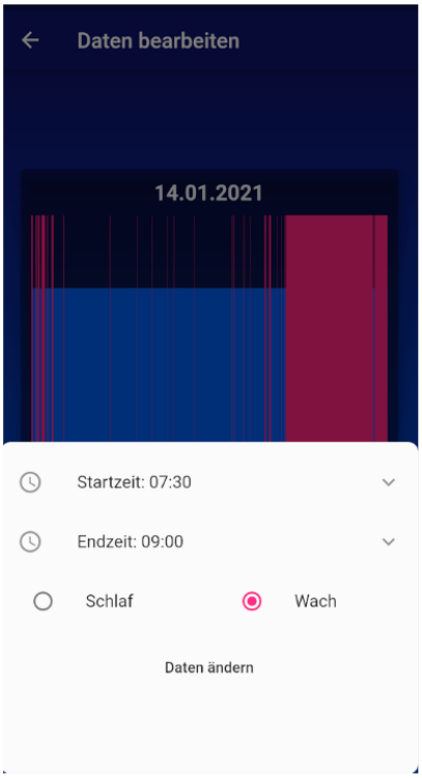

Abbildung 14: `ModalBottomSheet` zur Bearbeitung von Daten im `EditDetailsScreen`

2.3.8 Weitere Screens und Widgets

Außer den bisher vorgestellten Screens und Widgets gibt es noch folgende weitere:

- `ActivityScreen`:
  In diesem Screen soll eine Auswertung der vom Nutzer angegebenen Aktivitäten und Medikamente in Hinblick auf deren Einfluss auf den Schlaf erfolgen. Bisher ist dieser Screen jedoch nur *gemockt*, indem hier ein für die geplante Implementierung repräsentatives Bild eingebunden ist.

- `QuestionScreen`:
  In diesem Screen soll der Nutzer die Möglichkeit erhalten, verschiedene Fragebögen der Schlafdiagnostik auszufüllen, um einen tieferen Einblick in sein Schlafverhalten und Schlafmuster zu erhalten. Bisher ist dieser Screen jedoch nur *gemockt*, indem hier ein für die geplante Implementierung repräsentatives Bild eingebunden ist.

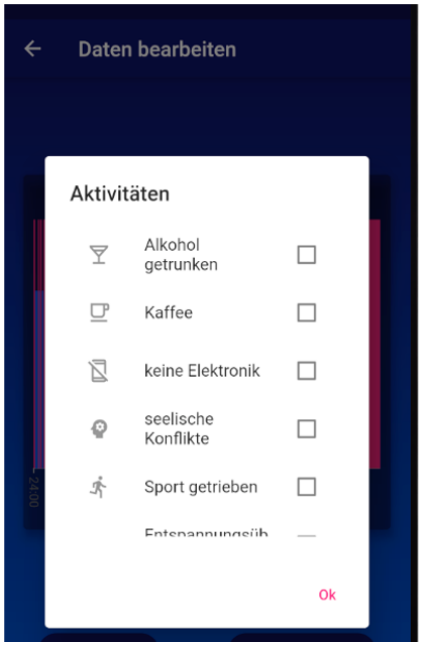

Abbildung 15: `AddActivities` Widget

- `MainDrawer` Widget:
  Dieses Widget, ist in die `AppBar` eingebunden und ermöglicht die Navigation zum `DisclaimerScreen`, zum `TutorialScreen`,
  zum `ConnectDeviceScreen` und zum `QuestionScreen`.

2.3.9 Foreground Service

Das Smartphone-Betriebssystem Android hat die Eigenschaft, dass Apps, die der Nutzer verlässt, aber nicht schließt, automatisch nach einer gewissen Zeit beendet werden. Da die Accelerometer-Daten in der Somnus-App jedoch permanent vom Fitness-Tracker gelesen werden müssen, darf die App vom Betriebssystem nicht abgeschaltet werden. Um das zu verhindern, wird direkt nach dem Start der App ein Foreground Service gestartet. Dieser platziert ein Banner in der Notification-Leiste von Android und stellt sicher, dass die App weiterhin Code ausführen kann, auch wenn die App nicht aktiv vom Nutzer verwendet wird.

2.4 Ordner `providers`

In diesem Ordner sind die Dateien enthalten, welche entweder Modelle für Datentypen enthalten oder Funktionen zur Kommunikation mit der Datenbank aufrufen.

2.5 Ordner `test`

In diesem Ordner sind Testdateien zu den jeweiligen Screens enthalten. Die Screens der Flutter-App wurden mit Widget-Tests getestet. Dabei wurden die Hauptelemente der Screens untersucht und auf korrekte Darstellung geprüft.

2.6 Verwendete Packages

Folgende Packages von wurden im Rahmen der Implementierung verwendet: 16

```
dependencies:
  flutter:
    sdk: flutter
  flutter_ble_lib: ^2.3.0
  permission_handler: ^5.0.1+1
  pointycastle: ^2.0.0
  custom_navigator: ^0.3.0
  introduction_screen: ^1.0.9
  shared_preferences: ^0.5.12+4
  flutter_localizations: # Add this line
    sdk: flutter
  syncfusion_flutter_charts: ^18.3.52
  intl: ^0.16.1
  provider: ^4.3.2+3
  pie_chart: ^4.0.1
  font_awesome_flutter: ^8.11.0
  printing: ^3.7.2
  pdf: ^1.13.0
  sqflite: ^1.3.0
  path_provider: ^1.6.24
  http: ^0.12.2
  foreground_service: ^2.0.1+1
  loading_overlay: ^0.2.1
  flutter_staggered_grid_view: ^0.3.3
```

Abbildung 16: Verwendete Packages

2.7 Datenbank

Als Datenbank wird SQLite verwendet. Die Datei „Database_Helper.dart“beinhaltet:

data model and constructor Grundsätzlich beinhaltet die Datenbank zwei Tabellen. Zum einen die „my_table “, welche die Werte des Aktivitätstrackers beinhalten und die „results_table “welche die erhaltenen Ergebnisse aus dem Backend speichert. Die Datenbank ist appweit erreichbar s. Listing 6.

Listing 6: data model and constructor

```
1
2  class DatabaseHelper {
3    static final _databaseName = "MyDatabasenew.db";
4    static final _databaseVersion = 1;
5
6    static final table = 'my_table';
7    static final results = 'results_table';
8
9    static final columnId = '_id';
10   static final columnName = 'name';
11   static final columnAge = 'age';
12   static final columnDate = 'date';
13   static final columnTime = 'time';
14   static final columnX = 'accx';
15   static final columnY = 'accy';
16   static final columnZ = 'accz';
17   static final columnT = 'acct';
18   static final columnSleepwake = 'sleepwake';
19
20   // make this a singleton class
21   DatabaseHelper._privateConstructor();
22   static final DatabaseHelper instance =
         DatabaseHelper._privateConstructor();
23
24   // only have a single app-wide reference to the database
25   static Database _database;
26   Future<Database> get database async {
27     if (_database != null) return _database;
28     // lazily instantiate the db the first time it is accessed
29     _database = await _initDatabase();
30     return _database;
31   }
```

open database Öffnen der Datenbank:

Listing 7: open database

```
// this opens the database (and creates it if it doesn't exist)
 _initDatabase() async {
   Directory documentsDirectory = await
       getApplicationDocumentsDirectory();
   String path = join(documentsDirectory.path, _databaseName);
   //print('db location : ' + path);
   return await openDatabase(path,
       version: _databaseVersion, onCreate: _onCreate);
 }
```

data tables Es werden zwei Tabellen benötigt, die „table "enthält die aufgezeichneten Accelerometerdaten des Mibands. Die Tabelle „results "speichert die vom Backend erhaltenen Auswertungen, welche vom Nutzer nachträglich noch bearbeitet werden können. Erstellen der zwei Datenbanktabellen :

```
// SQL code to create the database table
 Future _onCreate(Database db, int version) async {
   await db.execute('''CREATE TABLE $table (
    $columnId INTEGER PRIMARY KEY,
    $columnDate TEXT NOT NULL,
    $columnTime TEXT NOT NULL,
    $columnX REAL NOT NULL,
    $columnY REAL NOT NULL,
    $columnZ REAL NOT NULL,
    $columnT REAL NOT NULL
    )
    ''');
    await db.execute('''
    create table $results (
    $columnId INTEGER PRIMARY KEY,
    $columnDate TEXT NOT NULL,
    $columnTime TEXT NOT NULL,
    $columnSleepwake DOUBLE NOT NULL
    )''');
 }
```

Listing 8: open database

database helper functions Anschließend werden die Hilfsfunktionen definiert. Diese können überall in den einzelnen Widgets der App verwendet werden bei und wurden an dieser Stelle einmalig definiert.

```
// Helper methods

  // Inserts a row in the database where each key in the Map is a
    column name and the value is the column value. The return value
    is the id of the inserted row.
  Future<int> insert(Map<String, dynamic> row) async {
    Database db = await instance.database;
    return await db.insert(table, row);
  }

  Future<int> insertsleepwake(Map<String, dynamic> row) async {
    Database db = await instance.database;
    return await db.insert(results, row);
  }
```

Listing 9: fill database tables

3 Fitness-Armband

Das Schlafverhalten einer Person kann heutzutage sehr genau mithilfe eines Fitness-Armbands analysiert werden. Dafür werden lediglich Art und Häufigkeit der Bewegungen der Person erfasst und mithilfe eines Algorithmus ausgewertet. Fitness-Armbänder können bereits ab 30 Euro erworben werden. Mit dem günstigen Preis und der verfügbaren Funktionalitäten stellten Fitness-Armbänder für dieses Projekt eine wichtige Grundlage dar.

Da die Fitness-Armbänder jedoch kommerziell mit einer speziellen App vertrieben werden, ist die Software sowohl auf Hardware-, als auch auf Software-Seite Closed-Source. Zwar ist die zugrundeliegende Technologie in jedem Fall Bluetooth Low Energy, aber die genauen Protokolle zur Datenübertragung zwischen Armband und Smartphone-App sind nicht bekannt. Ein Vergleich diverser Fitness-Armbänder und eine ausführliche Recherche stellten heraus, dass einige Entwickler in der Community die Kommunikationsprotokolle des Fitness-Armbands MiBand 2 von Xiaomi herausfinden konnten und veröffentlichten. Da alle für das Somnus-Projekt benötigten Funktionen mit dem MiBand 2 realisiert werden konnten, wurde es für dieses Projekt ausgewählt.

3.1 Grundlagen

3.1.1 Bluetooth Low Energy

Bluetooth Low Energy (BLE) ist ein weitverbreiteter Funktechnologie-Standard von der Bluetooth Special Interest Group (Bluetooth SIG). Wegen der sehr energiesparenden Arbeitsweise, wird der Standard häufig in Wearables verwendet. Bei einer aufrechten Verbindung nimmt das Wearable die Rolle des Peripheral an und das Smartphone die Rolle des Central. Das Smartphone agiert als Master und kann Daten an den Slave senden bzw. vom Slave lesen. Aber auch der Slave kann unaufgefordert Daten an den Master senden, per Notification oder Indication (Notification mit Antwort). [Hey13]

BLE bietet bereits standardisierte Services und Charakteristiken, wie z.B. das Auslesen des Akkustandes oder das Senden eines Alarms. Die Hersteller des jeweiligen Geräts können aber auch eigene Services und Charakteristiken implementieren und anwendungsspezifische Daten übertragen. Ein Service ist eine Sammlung von Charakteristiken und eine Charakteristik beschreibt eine Interaktionsmöglichkeit mit einer Ressource. Services und Charakteristiken werden mit einem Universally Unique Identifier (UUID), der eine 128-bit große Zahl darstellt, gekennzeichnet [Hey13].

Beim MiBand 2 werden sowohl standardisierte, als auch eigene Services und Charakteristiken verwendet. Durch Recherche und Vergleich mit den standardisierten UUIDs konnten fast alle Services und Charakteristiken des MiBand 2 identifiziert werden. Eine ausführliche Liste befindet sich im Anhang (s. Tabelle 3).

3.1.2 Accelerometer

Ein Accelerometer ist ein Sensor, der Beschleunigungen misst. Die meisten Accelerometer messen die Beschleunigung auf drei Achsen (x, y und z), jedoch existieren auch Modelle, die weniger oder mehr Achsen verwenden. Beschleunigungen werden in der Regel in der Einheit m/s^2 angegeben. Bei Accelerometer-Daten wird jedoch die Einheit g für Gravitation eingesetzt. Der Wert 1 g entsprechen dann 9,8 m/s^2. Ist der Accelerometer wie in Abbildung 17 ausgerichtet, misst der Sensor im Ruhezustand 0 g auf der X- und Y-Achse, und auf der Z-Achse -1 g, da diese in entgegengesetzter Richtung zur Erdanziehungskraft ausgerichtet ist. Bei Bewegung verändern sich die gemessenen Werte auf den einzelnen Achsen und können Werte größer bzw. kleiner 1 annehmen. Der Wertebereich der Achsen ist variabel und abhängig von der Empfindlichkeit des Sensors. [Dej]

3.2 Verbindungsaufbau und Authentifizierung

Im Code für das Frontend wurde die Singleton-Klasse `BleDeviceController` angelegt, die die gesamte Bluetooth Low Energy Steuerung übernehmen sollte. Hier wurde die Verbindung mit dem MiBand 2 hergestellt, eine Authentifizierung durchgeführt und Schreib- und Lesebefehle auf die Charakteristiken durchgeführt. Für die BLE-Kommunikation wird das Package `flutter_ble_lib` verwendet.

Nach einem erfolgreichen Verbindungsaufbau mit dem MiBand 2, muss sich das Smartphone authentifizieren. Die benötigten Schritte für den Authentifizierungsprozess wurden detailliert von Andrey Nikishaev auf medium.com veröffentlicht [Nik18]. Zwischen Smartphone und MiBand 2 müssen einige Daten ausgetauscht werden, wobei nach jedem Schreibbefehl auf das MiBand 2, eine Antwort als Notification zurückkommt. Die Handler-Methode, die die Antworten interpretiert ist im Listing 10 dargestellt.

Im Somnus-Projekt wird der Authentifizierungsprozess in der Methode `authenticateMiBand` gestartet und beginnt mit dem Senden des 16 Bytes langen geheimen Schlüssels an das MiBand 2. Ist der Austausch des Schlüssels erfolgreich, fragt das Smartphone eine zufällige Zahl vom MiBand an. Die zufällige Zahl wird mit der AES-Verschlüsselung verschlüsselt und anschließend wieder an das MiBand 2 gesendet. Das MiBand 2 prüft den

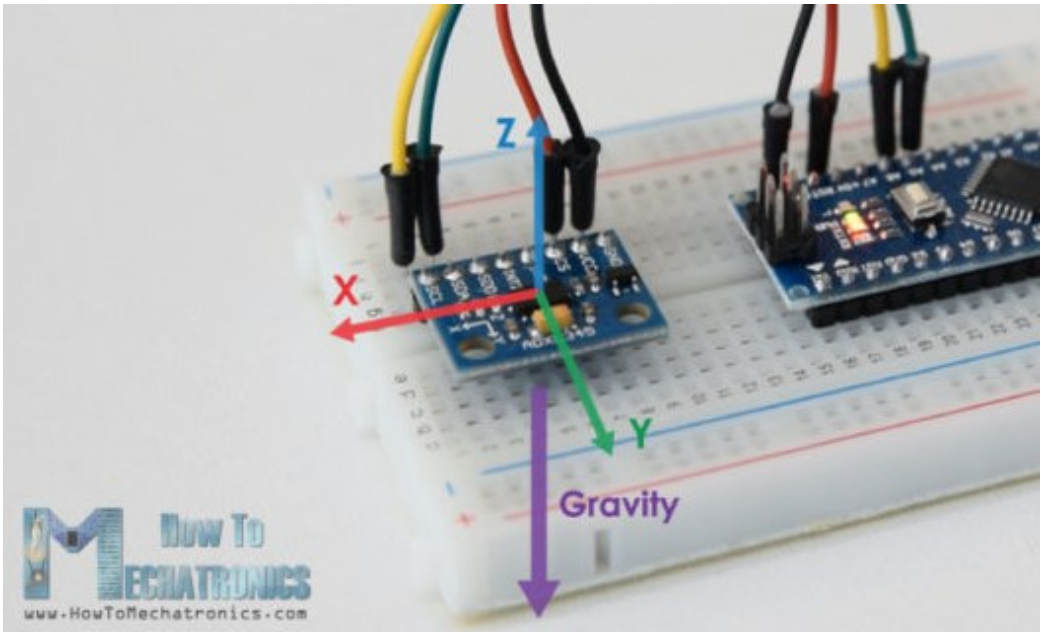

Abbildung 17: Ein Accelerometer mit eingezeichneten Achsen. [Dej]

verschlüsselten Inhalt und akzeptiert die Verbindung oder lehnt sie ab (siehe Listing 10).

```
Future<void> _handleAuthNotification(Uint8List data) async {
    if (data[0] == 16 && data[1] == 1 && data[2] == 1) {
        await _requestRand();
    } else if (data[0] == 16 && data[1] == 1 && data[2] == 4) {
        print("Error - Key sending failed.");
        _authenticationProcessFinish(false);
    } else if (data[0] == 16 && data[1] == 2 && data[2] == 1) {
        await _sendEncrRand(data.sublist(3));
    } else if (data[0] == 16 && data[1] == 2 && data[2] == 4) {
        print("Error - Request random number failed.");
        _authenticationProcessFinish(false);
    } else if (data[0] == 16 && data[1] == 3 && data[2] == 1) {
        print("AUTHENTICATED!!!");
        ForegroundService.notification.setText(DEVICE_CONNECTED);
        _authenticationProcessFinish(true);
        startReceivingRawSensorData();
    } else if (data[0] == 16 && data[1] == 3 && data[2] == 4) {
        print("Error - Encryption failed.");
        _authenticationProcessFinish(false);
    } else {
        print("Error - Authentication failed for unknown reason.");
        _authenticationProcessFinish(false);
    }
}
```

Listing 10: Notification-Handler-Methode für den Authentifizierungsprozess

3.3 Auslesen der Accelerometer-Daten

Für das Auslesen von Accelerometer-Werten stellte BLE keine standardisierte Charakteristik zur Verfügung. Auch von Xiaomi existierte keine offizielle Dokumentation, wie die Daten von dem Sensor ausgelesen werden konnten. In einem GitHub-Issue wurde jedoch ein Post gefunden, der eine Vorgehensweise zum Auslesen der Rohdaten beschreibte (siehe [rag19]). Die Vorgehensweise kann grob in zwei Schritte geteilt werden. Im ersten Schritt müssen zwei Werte auf die Sensor-Charakteristik mit der UUID 00000001-0000-3512-2118-0009af100700 geschrieben werden (Methode `_enableSendingRawSensorData`). Damit wird der Prozess zum Senden der Accelerometer-Rohdaten auf dem Mi-Band 2 angestoßen. Im zweiten Schritt wird die Accelerometer-Charakteristik mit der UUID 00000002-0000-3512-2118-0009af100700 auf Notifications be-

lauscht. Über diese Notification sendet das MiBand 2 die Accelerometer-Rohdaten an das Smartphone.

Im Post wird darauf hingewiesen, dass nach ca. 1:10 min keine Rohdaten mehr gesendet werden. Dies wurde durch eigene Beobachtungen verifiziert. Dieses Problem kann umgangen werden, indem der erste Schritt z.B. alle 30 s wiederholt wird. Wie sich jedoch gezeigt hat, kommt es beim erneuten Ausführen von Schritt 1 zu einer Pause von zwei Sekunden, in denen keine Rohdaten gesendet werden.

3.4 Interpretation der Accelerometer-Daten

Über die Notification werden im Durchschnitt ca. 10 Pakete pro Sekunde empfangen. Ein Paket kann dabei 8, 14 oder 20 Bytes beinhalten.

3.4.1 Paketstruktur

Im Post auf das GitHub-Issue wird erläutert, wie der Paketinhalt interpretiert werden kann. Aus Konventionsgründen werden die Paketdaten im Folgenden in hexadezimaler Darstellung beschrieben. In Tabelle 1 ist beispielhaft ein vom MiBand 2 gesendetes Paket dargestellt.

Byte 1
: Das erste Byte vom Paket ist immer 0x01.

Byte 2
: Das zweite Byte ist der Paketcounter. Er kann Werte von 0x00 bis 0xFF annehmen. Ein Paket wird vom Band immer mehrmals gesendet, deshalb wird der Paketcounter als Filter verwendet. Wird der maximale Werte erreicht, beginnt der Counter wieder bei 0x00.

Byte 3 - 8
: Diese Bytes sind die Accelerometer-Rohdaten. Das 3. und 4. Byte entsprechen dem X-Wert, das 5. und 6. Byte dem Y-Wert und das 7. und 8. Byte dem Z-Wert des Sensors. Das erste Byte der jeweiligen Achse ist der Wert, während das zweite Byte das Vorzeichen codiert.

Byte	1	2	3	4	5	6	7	8	..20
Wert	0x01	0x02	0x1F	0x00	0xA0	0xFF	0xD1	0x00	...

Tabelle 1: Beispiel eines Accelerometer-Daten-Pakets, das vom MiBand 2 an das Central-Device per Notification gesendet wird

Hält das zweite Byte den Wert 0x00, ist das Vorzeichen positiv. Hält das zweite Byte den Wert 0xFF, ist das Vorzeichen negativ.

Byte 9 - 20
Teilweise sendet das MiBand 2 auch zusätzliche 6 oder 12 Bytes. Diese sind dann weitere Accelerometer-Rohdaten, in der gleichen Struktur wie bereits die Bytes 3 - 8.

3.4.2 Umrechnung der Rohdaten

Wie ein Accelerometer-Datenpaket vom MiBand 2 aufgebaut ist, hat der Post des GitHub-Users ragcsalo [rag19] sehr gut beschrieben. Über die Interpretation der empfangenen Bytes 3, 5 und 7, die den Wert der Achsen codierten, wurde in dem Beitrag jedoch nichts erwähnt. Aus diesem Grund war der erste intuitive Ansatz eine Umrechnung des erhaltenen Wertes auf eine Skala von 0 bis 1. Dieser stellte sich jedoch als falsch heraus. Deshalb musste durch eigene Beobachtungen der Rohdaten eine zutreffende Formel entwickelt werden. Dafür wurde das MiBand 2 in diverse ruhende Positionen versetzt und die empfangenen Rohdaten aufgezeichnet. Mit Hilfe der Aufzeichnungen wurde ein Muster deutlich und es konnte eine Umrechnungstabelle erzeugt werden (s. Tabelle 2).

Achswert	**Vorzeichen**	**Dezimalwert**
0	0	0
1	0	0,0078125
2	0	0,015625
15	0	0,1171875
32	0	0,25
64	0	0,5
96	0	0,75
128	0	1
255	255	-0,0078125
254	255	-0,015625
241	255	-0,1171875
224	255	-0,25
192	255	-0,5
160	255	-0,75
128	255	-1

Tabelle 2: Umrechnungstabelle der vom MiBand 2 gesendeten Rohdaten

Aus der Umrechnungstabelle wurde schließlich folgende Formel entwickelt, um den Wert einer Achse zu berechnen.

$$x_{\text{real}} = \frac{x_{\text{codiert}}}{255} - (\frac{2 \cdot x_{\text{Vorzeichen}}}{255})$$

Die Variable x_{codiert} steht für das erste der zwei Bytes, die einen Achswert codieren. Die Variable $x_{\text{Vorzeichen}}$ steht für das zweite Byte und kann nur die zwei Werte 0 oder 255 annehmen. Die Lösung der Rechnung ergibt dann den Achswert x_{real}.

4 Backend

Das Backend für die Somnus App ist nicht lokal auf dem Smartphone des Benutzers, sondern läuft auf einem externen Server. Die Kommunikation mit der App auf dem Smartphone läuft über eine REST-Schnittstelle. Das gesamte Backend ist in der Programmiersprache Python gehalten.

4.1 REST-Server

Um mit der Smartphone-App über eine REST-Schnittstelle kommunizieren zu können besitzt das Backend einen REST-Server. Dieser wird mit dem Python Web-Framework FLASK[1] realisiert.

Unser Server hat eine REST-Schnittstelle:

```
1 @app.route('/data', methods=['GET', 'POST'])
```

Die Smartphone-App an diese Schnittstelle mit der HTTP-Methode *POST* eine CSV-Datei mit den Accelerometer-Daten und bekommt als Rückgabe eine CSV-Datei mit der Schlaf-Analyse zurück.

[1] https://flask.palletsprojects.com/en/1.1.x/

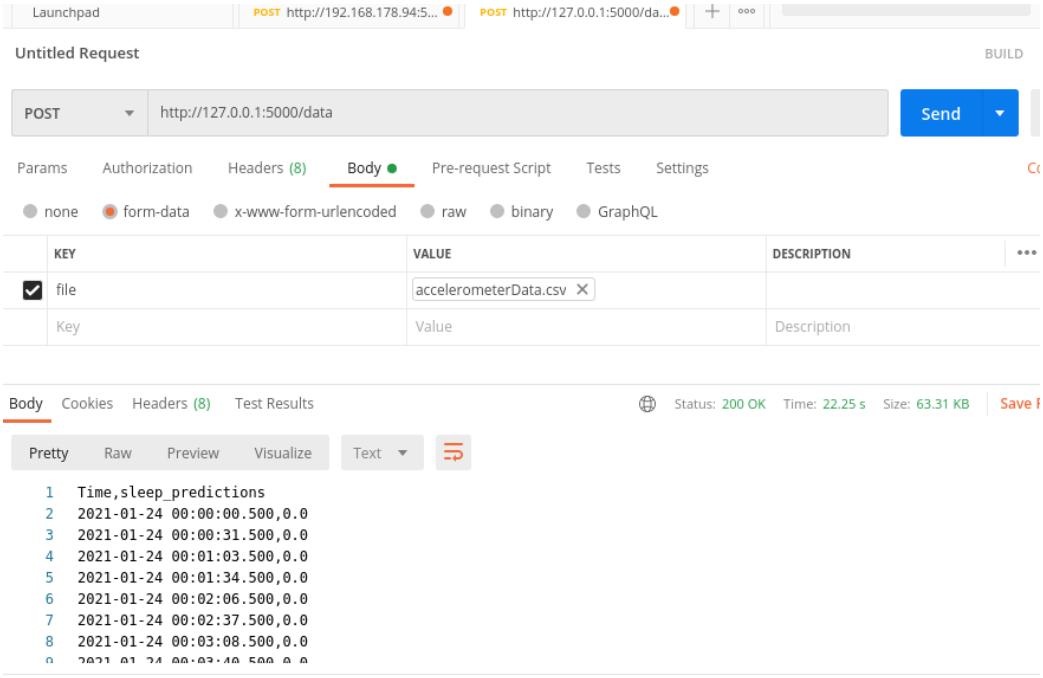

Abbildung 18: Ansprechen der Schnittstelle über Postman

upload_file() Bei Ansprechen der */data* Schnittstelle wird die im Server definierte Methode *upload_file():* ausgeführt. In ihr wird abgeprüft, ob eine Datei mit übertragen wurde und ob die Datei die Dateiendung .csv hat. Dann wird ein aktueller Zeitstempel erstellt und zusammen mit diesem ein neuer einzigartiger Name für die hochgeladene Datei erstellt:

```
timestamp = str(time.time())
newfilename = timestamp + 'fileUpload.csv'
```

Am Schluss wird die hochgeladene CSV-Datei auf dem Filesystem des Servers gespeichert und die im Server definierte Methode *run_data(src, stamp):* wird ausgeführt.

run_data() In die Server-Datei wird unser eigenes Modul *Somnus* importiert. in der Server-Methode *run_data()* wird nun die *run*-Methode des Moduls Somnus ausgeführt und als Parameter der Pfad zu der CSV-Datei übergeben:

```
def run_data(src, stamp):
    src = PROJECT_ROOT + '/fileUploads/' + src
    try:
        somnus.Somnus(input_file=src, sampling_frequency=100, verbose=True
    )
    except Exception as e:
```

```
        print("Error processing: {}\nError: {}".format(src, e))
    return send_file('results/' + stamp + 'fileUpload.csv')
```

Wenn die Schlafanalyse durchgelaufen ist wird eine CSV mit den Ergebnissen erzeugt und auf dem Filesystem des Servers abgelegt und dann über die REST-Schnittstelle zurück an die Smartphone-App geschickt.

clearbackend() Am Schluss wird die Server-Methode *clearbackend()* ausgeführt. Sie löscht alle CSV-Dateien aus dem Uploads-Ordner und alle CSV-Dateien aus dem Results-Ordner.

4.2 Somnus

Die Basis der Schlaf-/Wach-Bestimmung aus Accelerometer-Daten ist das GitHub-Projekt SleepPy[2] von Yiorgos Christakis, das wir nach unseren Anforderungen hin verändert, gekürzt und erweitert haben.

Vorbereitung

Zu Anfang wird die CSV-Datei mit den Accelerometer-Daten eingelesen und in eine Python Panda-Dataframe gespeichert (wie eine Tabelle). Dafür wird definiert, ab welcher Zeile die Accelerometer-Daten beginnen (falls Zusatzinformationen zu Beginn der CSV abgespeichert sind), welche Spalten der CSV benutzt werden und welchen Datentyp die Werte in den jeweiligen Spalten haben.

```
usecols=["Time", "X", "Y", "Z"],
dtype={ "Time": object,
        "X": np.float64,
        "Y": np.float64,
        "Z": np.float64},
```

Zum Schluss werden die Accelerometer-Daten in 24 Stunden Perioden unterteilt (falls die in der CSV-Datei enthaltenden Accelerometer-Daten einen Zeitraum von mehr als 24-Stunden umfassen) im HDF-Format in einem zuvor erstellten Ordner abgespeichert.

Aktivitätsindex

Die Funktion *extract_activity_index()* lädt die zuvor erstellte HDF-Datei ein und berechnet pro Zeile (also pro Zeitstempel anhand der X, Y, Z-Accelerometer-Daten einen Aktivitätsindex. Dafür benötigt sie einen Bandpass-

[2] https://github.com/elyiorgos/sleeppy/blob/master/sleeppy/sleep.py

Filter, der bei den Sensordaten nur ein bestimmtes Frequenzband passieren lässt.

```
critical_frequency = [
        bp_cutoff[0] * 2.0 / sampling_rate,
        bp_cutoff[1] * 2.0 / sampling_rate,
    ]
```

Zum Schluss wird eine HDF-Datei erstellt und abgespeichert, in der für jeden Zeitstempel einen Aktivitätsindex zugeordnet wird.

Schlaf-/Wach-Analyse

Herzstück unseres Programms ist der Cole-Kripke-Algorithmus [Col+92]. Er gehört zu den Algorithmen, die Actimetriedaten des Handgelenks auswerten, um Schlaf-/Wachphasen zu unterscheiden. Ein anderer häufig benutzter ist der Sadeh-Algorithmus [SSC94]. Studien haben gezeigt, dass zur Unterscheidung von Schlaf-Wachphasen Handgelenk-Actimetriedaten nicht viel schlechter sind als Polysomnographiedaten aus dem Schlaflabor [Mel+12] [Qua+18]. Für den Cole-Kripke-Algorithmus haben wir uns entschieden, weil er sich in einer Studie von Mirja Quante et al. im Vergleich zu Polysomographiedaten als akkurater herausgestellt hat als der Sadeh-Algorithmus [Qua+18].

In der Funktion *sleep_wake_predict(self):* wird die zuvor erstellte HDF-Datei eingelesen. Dann wird auf jeden Eintrag (bestehend aus Zeitstempeln und seinem Aktivitätsindex) die Funktionen der Python-Klasse *ColeKripke* angewendet: *predict* und *rescore*.

predict() Auf Basis des Aktivitätsindex wird festgelegt, ob der Mensch, von dem die Accelerometerdaten stammen, zu dem Zeitpunkt wach war oder geschlafen hat. Dafür wird der Funktion als Parameter ein Scale Factor übergeben, bei uns hat er den Wert 0.193125. Durch mathematische Faltung (Python NumPy-Funktion *convolve()*) mit dem Scale Factor entstehen Score-Werte. Diese Score-Werte werden mit dem Schwellwert 0.5 verglichen: Werte ≥ 0.5 werden zu 1 (Wach), Werte < 0.5 werden zu 0 (Schlaf).

Leider entsteht bei *predict()* ein Problem: Die Funktion missinterpretiert die Zeit, in der der Mensch, der das Accelerometer-Armband trägt schon ganz ruhig wird, aber noch nicht schläft fälschlicherweise als Schlaf. Um dieses Problem zu umgehen hat die Klasse *ColeKripke* eine weitere Funktion:

rescore() Auf Basis der 5 Rescoring Rules nach Webster[Web+82] werden die zuvor getroffenen Score-Werte noch einmal evaluiert und angepasst. Die fünf Regeln sind:

- 1. Regel: Nach einer Phase von mindestens 4 Minuten Wach wird die nächste Minute Schlaf als Wach rescored.
- 2. Regel: Nach einer Phase von mindestens 10 Minuten Wach werden die nächsten 3 Minuten Schlaf als Wach rescored.
- 3. Regel: Nach einer Phase von mindestens 15 Minuten Wach werden die nächsten 4 Minuten Schlaf als Wach rescored.
- 4. Regel: Gibt es eine Phase von 6 oder weniger Minuten Schlaf, die umgeben ist von mindestens 10 Minuten Wach, wird diese Phase als Wach rescored.
- 5. Regel: Gibt es eine Phase von 10 oder weniger Minuten Schlaf, die umgeben ist von mindestens 20 Minuten Wach, wird diese Phase als Wach rescored.

Zum Schluss werden die Ergebnisse in Form einer CSV abgespeichert. Pro Zeile gibt es den Zeitstempel und den Wert 1 für Wach und den Wert 0 für Schlaf.

Nachbereitung

In der Funktion *clear_data(self)* werden zum Schluss alle nicht mehr benötigten Verzeichnisse und HDF-Dateien gelöscht. Übrig bleibt nur eine CSV-Datei mit den Schlafvorhersagen pro 24-Stunden-Abschnitt.

Ausführung

Bei Anwendung unseres Python Skriptes wird die Funktion *run()* aufgerufen. In ihr werden nacheinander die oben genannten Schritte ausgeführt und während der jeweiligen Schritte wird eine Konsolenausgabe definiert, die den jeweiligen Status angibt.

```
def run(self):
    print('Sleep Detection gestartet')
    try:
        rmtree(self.sub_dst)  # removes old files from result directory.
        print('emptied result directory')
    except OSError:
        print('result was already empty')
```

```
8       os.mkdir(self.sub_dst)  # set up output directory
9       if self.verbose:
10          print("Loading CSV data, split data into 24 hour periods...")
11      self.split_days_csv()
12      if self.verbose:
13          print("Extracting activity index from accelerometer data...")
14      self.extract_activity_index()
15      if self.verbose:
16          print("Running sleep/wake predictions...")
17      self.sleep_wake_predict()
18      if self.verbose:
19          print("Clearing intermediate data...")
20      self.clear_data()
```

Listing 11: Methode run

4.3 Testing

Für die Integration-Tests wurde das Framework Pytest verwendet. Es gibt aktuell vier Tests. Zum Testen der Schnittstelle muss zunächst ein Testclient angelegt werden, welcher die Anfragen an die Schnittstelle simulieren kann. Anschließend werden folgende Szenarien getestet:

- Client stellt Anfrage mit Post und korrekter CSV
- Client stellt Anfrage mit falschem Dateityp
- Client stellt Anfrage ohne Dateiübergabe
- Client stellt Anfrage mit Key File aber ohne Datei

```
# create a test client
@pytest.fixture(scope='module')
def test_client():
    flask_app = server.app
    testing_client = flask_app.test_client()
    ctx = flask_app.app_context()
    ctx.push()
    yield testing_client
    ctx.pop()
```

```

#test with right csv
def test_file_upload(test_client):
    data = {
        'field': 'value',
        'file': ('tests/test.csv', 'test.csv')
    }

    rv = test_client.post('/data', buffered=True,
                          content_type='multipart/form-data',
                          data=data)
    assert rv.status_code == 200

#test errorhandling with wrong filetype
def test_wrong_filetype(test_client):
    data = {
        'field': 'value',
        'file': ('tests/test.png', 'test.png')
    }

    rv = test_client.post('/data', buffered=True,
                          content_type='multipart/form-data',
                          data=data)
```

```
24      assert rv.status_code == 415
25
26
27  #test errorhandling when no file is selected
28  def test_no_file(test_client):
29      data = {
30      }
31
32      rv = test_client.post('/data', buffered=True,
33                            content_type='multipart/form-data',
34                            data=data)
35      assert rv.status_code == 412
36
37
38  #test if key 'file' is selected, but no file attached
39  def test_file_empty(test_client):
40      data = {
41          'field': 'value',
42          'file': ''
43      }
44
45      rv = test_client.post('/data', buffered=True,
46                            content_type='multipart/form-data',
47                            data=data)
48      assert rv.status_code == 412
```

5 Literaturverzeichnis

[Col+92] Roger J Cole u. a. „Automatic sleep/wake identification from wrist activity“. In: *Sleep* 15.5 (1992), S. 461–469.

[Dej] Dejan. *How To Track Orientation with Arduino and ADXL345 Accelerometer*. URL: https://howtomechatronics.com/tutorials/arduino/how-to-track-orientation-with-arduino-and-adxl345-accelerometer/.

[Hey13] Robin Heydon. *Bluetooth Low Energy: The Developer's Handbook*. Prentice Hall, 2013.

[Mel+12] Lisa J Meltzer u. a. „Direct comparison of two new actigraphs and polysomnography in children and adolescents“. In: *Sleep* 35.1 (2012), S. 159–166.

[Nik18] Andrey Nikishaev. *How I hacked my Xiaomi MiBand 2 fitness tracker — a step-by-step Linux guide*. 26. Mai 2018. URL: https://medium.com/machine-learning-world/how-i-hacked-xiaomi-miband-2-to-control-it-from-linux-a5bd2f36d3ad (aufgerufen am 25. 01. 2021).

[Qua+18] Mirja Quante u. a. „Actigraphy-based sleep estimation in adolescents and adults: a comparison with polysomnography using two scoring algorithms“. In: *Nature and science of sleep* 10 (2018), S. 13.

[rag19] ragcsalo. 19. Mai 2019. URL: https://github.com/Freeyourgadget/Gadgetbridge/issues/63#issuecomment-493740447 (aufgerufen am 25. 01. 2021).

[SSC94] Avi Sadeh, M Sharkey und Mary A Carskadon. „Activity-based sleep-wake identification: an empirical test of methodological issues“. In: *Sleep* 17.3 (1994), S. 201–207.

[Web+82] John B Webster u. a. „An activity-based sleep monitor system for ambulatory use“. In: *Sleep* 5.4 (1982), S. 389–399.

A Anhang

Service UUID **- Charakteristik UUID**	**Bezeichnung**
00001800-0000-1000-8000-00805f9b34fb	Generic Access
- 00002a00-0000-1000-8000-00805f9b34fb	Device Name
- 00002a01-0000-1000-8000-00805f9b34fb	Appearance
- 00002a04-0000-1000-8000-00805f9b34fb	Peripheral Preferred Connection Parameters
00001801-0000-1000-8000-00805f9b34fb	Generic Attribute
- 00002a05-0000-1000-8000-00805f9b34fb	Service Changed
0000180a-0000-1000-8000-00805f9b34fb	Device Information
- 00002a25-0000-1000-8000-00805f9b34fb	Serial Number String
- 00002a27-0000-1000-8000-00805f9b34fb	Hardware Revision String
- 00002a28-0000-1000-8000-00805f9b34fb	Software Revision String
- 00002a23-0000-1000-8000-00805f9b34fb	System ID
- 00002a50-0000-1000-8000-00805f9b34fb	PnP ID
00001530-0000-3512-2118-0009af100700	Weight Service (Custom)
- 00001531-0000-3512-2118-0009af100700	N/A
- 00001532-0000-3512-2118-0009af100700	N/A
00001811-0000-1000-8000-00805f9b34fb	Alert Notification
- 00002a46-0000-1000-8000-00805f9b34fb	New Alert
- 00002a44-0000-1000-8000-00805f9b34fb	Alert Notification Control Point
00001802-0000-1000-8000-00805f9b34fb	Immediate Alert
- 00002a06-0000-1000-8000-00805f9b34fb	Alert Level
0000180d-0000-1000-8000-00805f9b34fb	Heart Rate
- 00002a37-0000-1000-8000-00805f9b34fb	Heart Rate Measurement
- 00002a39-0000-1000-8000-00805f9b34fb	Heart Rate Control Point
0000fee0-0000-1000-8000-00805f9b34fb	MiBand 0 (Custom Service)
- 00002a2b-0000-1000-8000-00805f9b34fb	Current Time
- 00000020-0000-3512-2118-0009af100700	N/A
- 00000001-0000-3512-2118-0009af100700	Sensors (Heart Rate and Accelerometer)
- 00000002-0000-3512-2118-0009af100700	Accelerometer
- 00000003-0000-3512-2118-0009af100700	Configuration
- 00002a04-0000-1000-8000-00805f9b34fb	Peripheral Preferred Connection Parameters
- 00000004-0000-3512-2118-0009af100700	Fetch
- 00000005-0000-3512-2118-0009af100700	Activity Data
- 00000006-0000-3512-2118-0009af100700	Battery

- 00000007-0000-3512-2118-0009af100700	Steps
- 00000008-0000-3512-2118-0009af100700	User Settings
- 00000010-0000-3512-2118-0009af100700	Device Event
0000fee1-0000-1000-8000-00805f9b34fb	MiBand 1 (Custom Service)
- 00000009-0000-3512-2118-0009af100700	Authentication
- 0000fedd-0000-1000-8000-00805f9b34fb	Jawbone
- 0000fede-0000-1000-8000-00805f9b34fb	Coin, Inc.
- 0000fedf-0000-1000-8000-00805f9b34fb	Design SHIFT
- 0000fed0-0000-1000-8000-00805f9b34fb	Apple, Inc.
- 0000fed1-0000-1000-8000-00805f9b34fb	Apple, Inc.
- 0000fed2-0000-1000-8000-00805f9b34fb	Apple, Inc.
- 0000fed3-0000-1000-8000-00805f9b34fb	Apple, Inc.
- 0000fec1-0000-3512-2118-0009af100700	KDDI Corporation

Tabelle 3: Services und Charakteristiken des MiBand 2