Internet of Things for Industry and Human Applications

Mobile and Hybrid Internet of Things Based Computing

Internet of Things
for Industry and Human Applications

Mobile and Hybrid
Internet of Things Based Computing

PRACTICUM

Ministry of Education and Science of Ukraine
National Aerospace University "KhAI"

V.O. Butenko, O.N. Odarushchenko, A.Y. Strjuk,
E.B. Odarushchenko, D.A. Butenko

Mobile and hybrid Internet of Things based computing

Practicum

Edited by V. S. Kharchenko

Project
ERASMUS+ ALIOT "Modernization Internet of Things: Emerging Curriculum for Industry and Human Applications Domains" (573818-EPP-1-2016-1-UK-EPPKA2-CBHE-JP)

2019

UDC 004.382.74iOS _And:004.411](076.5)=111
MC77

Reviewers:
DrS, Prof. Volodymyr Mokhor, director of Pukhov Institute for Modelling in Energy Engineering, corresponding member of NAS of Ukraine
Dr. Ah-Lian Kor, Leeds Beckett University, UK

M77 Butenko V.O., Odarushchenko O.N., Strjuk A.Y., Odarushchenko E.B., **Mobile and hybrid Internet of Things based computing**: Practicum / Kharchenko V.S. (Ed.) – Ministry of Education and Science of Ukraine, National Aerospace University "KhAI", 2019. – 124 p.

ISBN 978-617-7361-87-8

The materials of the practical part of the master course "MC3. Mobile and hybrid IoT-based computing", developed in the framework of the ERASMUS+ ALIOT project "Modernization Internet of Things: Emerging Curriculum for Industry and Human Applications Domains" (573818-EPP-1-2016-1-UK-EPPKA2-CBHE-JP).

Study material presented in this practical part of the master course is covering the basic topics iOS and Android application development and there use for IoT systems.

It is intended for engineers, developers and scientists engaged in the development and implementation of of IoT-based systems, for postgraduate students of universities studying in areas of IoT, computer science, computer and software engineering, as well as for teachers of relevant courses.

Ref. – 38 items, figures – 66.

Approved by Academic Council of National Aerospace University "Kharkiv Aviation Institute" (record No 4, December 19, 2018).

ISBN 978-617-7361-87-8

© Butenko V.O., Odarushchenko O.N., Strjuk A.Y., Odarushchenko E.B., Butenko D.A.

This work is subject to copyright. All rights are reserved by the authors, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms, or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar.

Міністерство освіти і науки України
Національний аерокосмічний університет
ім. М. Є. Жуковського «Харківський Авіаційний Інститут”

В.О.Бутенко, О.М. Одарущенко, О.Ю. Стрюк,
Е.В. Одарущенко, Д.А. Бутенко

Мобільні і гібридні обчислення на основі Інтернету речей

Практикум

Редактор Харченко В.С.

Проект ERASMUS+ ALIOT
“Інтернет речей: нова освітня програма для потреб промисловості та суспільства”
(573818-EPP-1-2016-1-UK-EPPKA2-CBHE-JP)

2019

УДК004.382.74iOS_And:004.411](076.5)=111
М77
Рецензенти:

Д.т.н., проф. Володимир Мохор, директор інституту проблем моделювання в енергетиці ім. Г.Є. Пухова, член-кореспондент НАН України

Др. А-Ліан Кор, Leeds Beckett University, Велика Британія

М77 Бутенко В.О., Одарущенко О.М., Стрюк О.Ю., Одарущенко Е.В., Бутенко Д.А. **Мобільні і гібридні обчислення на основі Інтернету речей.** / За ред. Харченка В.С. – МОН України, Національний аерокосмічний університет ім. М. Є. Жуковського «ХАІ». – 124 с.

ISBN 978-617-7361-87-8

Викладено матеріали практичної частини курсу "MC3. Mobile and hybrid IoT-based computing", підготовленого в рамках проекту ERASMUS+ ALIOT "Internet of Things: Emerging Curriculum for Industry and Human Applications" (573818-EPP-1-2016-1-UK-EPPKA2-CBHE-JP).

Навчальний матеріал, представлений у цій практичній частині магістерського курсу, висвітлює основні теми розробки додатків для iOS та Android та використання їх для систем IoT.

Призначено для інженерів, розробників та науковців, які займаються розробкою та впровадженням IoT для промислових систем, для аспірантів університетів, які навчаються за напрямом комп'ютерних наук, комп'ютерної та програмної інженерії, а також для викладачів відповідних курсів.

Бібл. – 38, рисунків – 66.

Затверджено Вченою радою Національного аерокосмічного університету «Харківський авіаційний інститут» (запис № 4, грудень 19, 2018).

ISBN 978-617-7361-87-8

© Бутенко В.О., Одарущенко О.М., Стрюк О.Ю., Одарущенко Е.В., Бутенко Д.А.

Ця робота захищена авторським правом. Всі права зарезервовані авторами, незалежно від того, чи стосується це всього матеріалу або його частини, зокрема права на переклади на інші мови, перевидання, повторне використання ілюстрацій, декламацію, трансляцію, відтворення на мікрофільмах або будь-яким іншим фізичним способом, а також передачу, зберігання та електронну адаптацію за допомогою комп'ютерного програмного забезпечення в будь-якому вигляді, або ж аналогічним або іншим відомим способом, або ж таким, який буде розроблений в майбутньому.

ABBREVIATIONS

MC Master Course
API Application Programming Interface
IDE Integrated Development Environment
ID Identifier
BLE Bluetooth Low Energy
UUID Universally Unique Identifier
UI User Intrerface

INTRODUCTION

The materials of the practical part of the master course "MC3. Mobile and hybrid IoT-based computing", developed in the framework of the ERASMUS+ ALIOT project "Modernization Internet of Things: Emerging Curriculum for Industry and Human Applications Domains" (573818-EPP-1-2016-1-UK-EPPKA2-CBHE-JP)[1].

Study material presented in this practical part of the master course is covering the basic topics iOS and Android application development and there using for IoT systems.

The main topics of practical works are following:

- getting started with XCode and Android Studio - setting up your development environment;
- design and basic layouts of the iOSand Android diabetic tracer application "Glu";
- translating design into code - add and setup basic "Glu" components;
- getting started with data storages for iOS and Android;
- assessing user health information using HealthKit and Google Fit;
- Integrating third-party trackers and glucometers using API.

The course is intended for engineers, developers and scientists engaged in the development and implementation of IoT-based systems, for postgraduate students of universities studying in areas of IoT, computer science, computer and software engineering, as well as for teachers of relevant courses.

Practicum prepared by Dr. Butenko V.O., Dr. Odarushchenko O.M., Dr. Strjuk O.Y., Butenko D.A. (National Aerospace University "KhAI") and Dr. Odarushchenko O.B. (Poltava State Agrarian Academy). General editing was performed by DrS. Kharchenko V.S.

The authors are grateful to the reviewers, project colleagues, staff of the departments of academic universities, industrial partners for valuable information, methodological assistance and constructive suggestions that were made during the course program discussion and assistance materials.

[1] *The European Commission's support for the production of this publication does not constitute an endorsement of the contents, which reflect the views only of the authors, andthe Commission cannot be held responsible for any use which may be made of the information contained therein.*

1. Developing IoT-based application for iOS

Practical work 1.1

GETTING STARTED WITH XCODE – INTRODUCTION TO THE IDE

1.1.1 Synopsis

In this practical work we will discuss the Xcode basics – the native IDE for iOS, macOS, watchOS and tvOS. This practical work is a brief introduction to the Apple coding environment that aims to show basic functionality of the IDE by creating a playground and a single view application project, use git to make a version control and CocoaPods to scale project with various libraries.

1.1.2 Brief theoretical information

Xcode — is an integrated development environment (IDE) for software on macOS, iOS, watchOS and tvOS platforms developed by Apple.

Xcode provides developers with documentation and Interface Builder – imbedded application for graphic user interfaces construction. The Xcode bundle consist of following sources: XCode supports source code for the languages C, C++, Swift, Objective-C, Objective C++, Java, Python, Ruby and ResEdit with a variety of programming models, including but not limited to Cocoa, Carbon and Java. The third parties have added the support of GNU Pascal, Free Pascal, Ada, C#, Haskell, Perl and D. Xcode suite uses the LLDB debugger as the back-end for the IDE's debugger.

During this course we will use Swift to develop the glucose management application. Swift is a general-purpose, multi-paradigm, compiled programming language created by Apple for iOS, macOS, watchOS, tvOS, Linux and z/OS platforms.

While supporting the most Objective-C concepts such as dynamic dispatch, widespread late building, the Swift is intended to provide “safer” way to ease the software bugs catching. Swift supports the concept of protocol extensibility that can be applied to types, structures and classes etc.

The given practical works are based on Swift 4.2 presented in

2018 by Apple along with iOS12 [1].

1.1.3 Practical steps

The Xcode is free and can be downloaded via AppStore. Being multifunctional the Xcode have integrated iOS SDK package, code editor, Interface Builder, debugger, iPhone/iPad/Apple Watch/Apple TV simulators. The Xcode welcome screen is presented on Figure 1.1. Here we can create a new project, playground, clone existing project or navigate to the recent one.

Figure 1.1 – Xcode welcome screen

1. Creating a first playground

The playground is a perfect place to learn Swift, quickly write some code, experiment with new syntax or to test algorithms. Here we can test new code and immediately see the real-time results. The playground window consists of two basic parts: code editor in the left side and results view part on the right side (Figure 1.2).

Add following code to the playground (Code sample 1.1).

```
import UIKit

let frameView = CGRect(x: 0, y: 0, width:
150, height: 150)
let customView = UIView(frame: frameView)
customView.backgroundColor = UIColor.blue
```

Code sample 1.1

Figure 1.2

The first line imports a UIKit framework that construct and manage a graphical, event-driven user interface for iOS and tvOS applications. Next, we create a frame with CGRect(x:y:width:height) method and use this frame to initialize UIView. Lastly, change it color to blue.

2. Creating a first Xcode project.

Go back to the welcome screen and create a new Xcode project. In following window (Figure 1.3) navigate to the iOS tab. There are several project types that can be created by now:

- **Single view application** – is the most commonly selected application type. The template includes default UIViewController and it's class;
- **Game** – the template includes a GameViewController with basic files to develop gaming scenes (GameScene.sks, Actions.sks and GameScene.swift);
- **Augmented Reality App** – the template which contains a default UIViewController and it's class with imported ARKit and some methods from ARSCNViewDelegate;
- **Document Based App** – the template with the standard Document Browser View Controller and several default classes;
- **Master-Detail App** – the template includes a predefined views tree that incorporated Master View Controller with two Navigation Controllers, Table View Controller and UiViewController;
- **Page-Based App** – the template gives a default UIPageViewController with it's classes;
- **Tabbed App** – provides a template with UITabBarController

that have a segue to the two UiViewControllers along with their classes;

- **Sticker Pack App** – the template has a Stickers.xcstikers file for new stickers integration;
- **iMessage App** – the template gives a storyboard and controller to create iMessage extension.

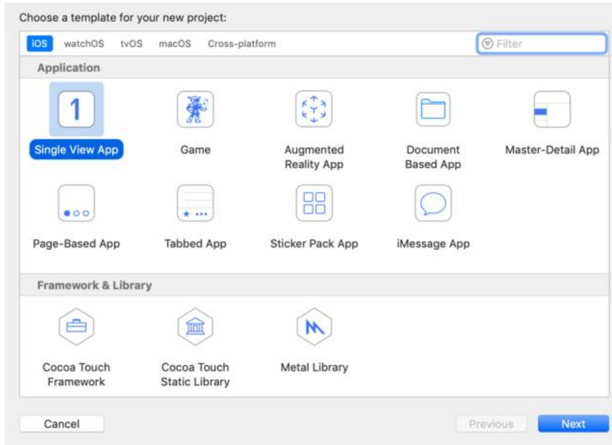


Figure 1.3

Select the Single View App type and in the next window enter following information (Figure 1.4):

- **Product name** – is a newly created project name, for example «HelloWorld»;
- **Team** – is an account name created on developer.apple.com;
- **Organization name** – is a developer or organization name. If nothing is defined the system will take the Mac account name;
- **Organization identifier** – mainly presents a domen organization name written in backwards order (com.name) that aims to make a unique application identifier;
- **Bundle identifier** – the unique application identifier that is created with project name and organization ID, for example «com.name.HelloWorld»;
- **Language** – is a project language (Swift or Objective-C). We need to select Swift.

On the next screen we need to select a destination folder tor

newly created project and create the Git repository if it is needed.

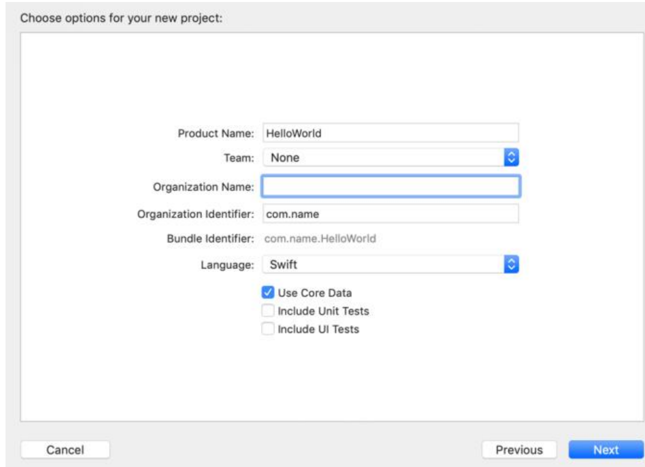


Figure 1.4

3. Xcode windows

The Figure 1.5 shows Xcode interface for the new «HelloWorld» single view application that contains the Navigation (1), Editor (2), Utility (3), Debug (4) and Toolbar (5) areas. We can change size of each window and hide or show them using buttons in the right top corner.

Let us discuss each Xcode component;

1) The **Navigation** gives bunch of tools that can help to navigate through project files a as well as build, debug and run stages. The first tab presents project files tree (Figure 1.6). The files groups can be created during project development, but it should be noted that creation of a new group inside of Xcode doesn't mean that folder with the same name was actually created in original project folder. To escape the complications during final project debug all project folders should be created manually and then added to Xcode with right button click on specified folder and in popup menu «Add new files to «HelloWorld»» selection. Using Search tool in Navigation bar we can easily look for interested text information (Figure 1.7). The Issue tab presents all problems that appear during compilation and helps to quickly jump to that place in code (Figure 1.8).

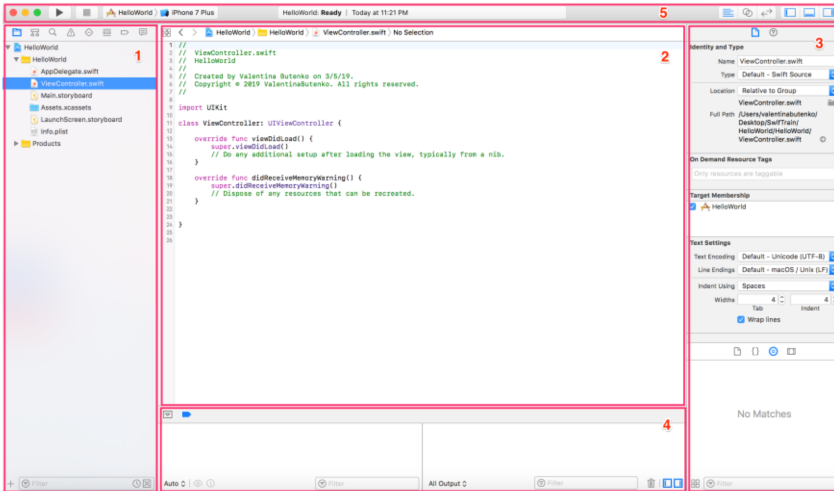
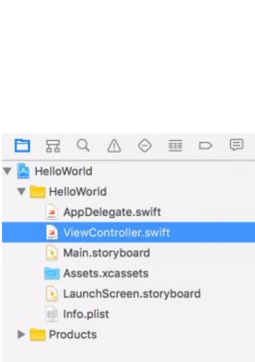
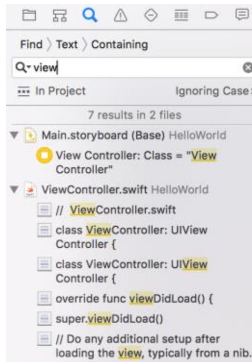


Figure 1.5





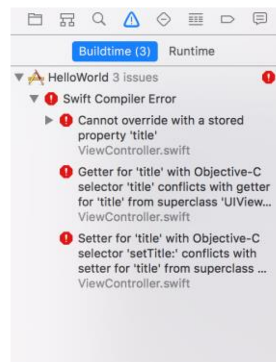


Figure 1.6 Figure 1.7 Figure 1.8

2) **Code Editor** is a place where developer spends main time during application development (Figure 1.9). The navigation through project files can be made using Navigation bar, as it was previously discussed, or in Code Editor options directly. Those options present the work files hierarchy and even give information on methods in each file (Figure 1.10).

The Editor window can be presented in three possible types: single

window (Figure 1.11), Version Editor to see the changes in file (Figure 1.12) and Assistant Editor Window that shows two work files (Figure 1.13).

Figure 1.9

Figure 1.10

Figure 1.11

Figure 1.12

Before we go to the next window there is one important thing that should be highlighted in Editor Window – breakpoint. The breakpoints are heavily used during code debug and mark the line to pause an application as soon as it enters that line. Breakpoint are simply placed by clicking on the number of needed line (Figure 1.14)

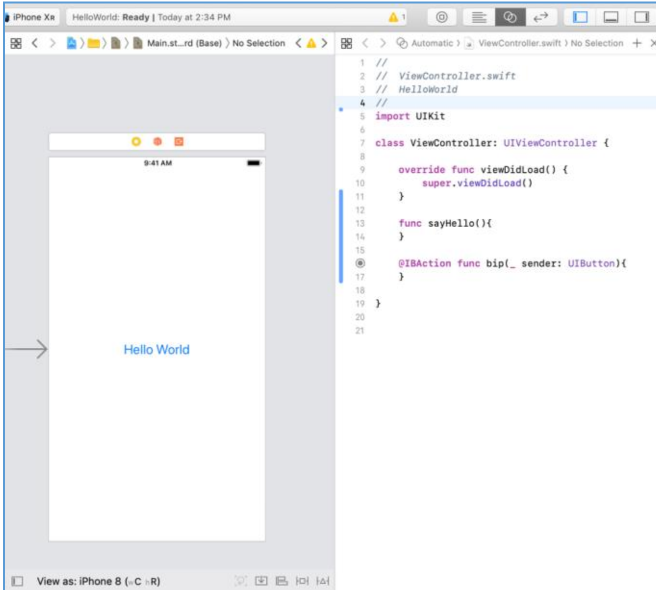


Figure 1.13

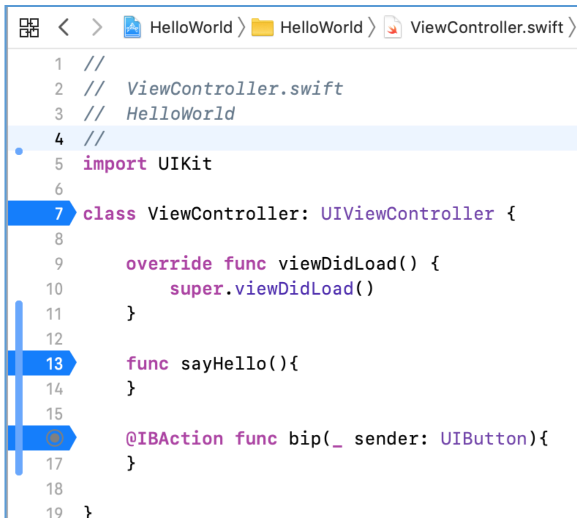


Figure 1.14

3) **Utility** window. One of the most frequently used Utility tabs is an Attributes Inspector (Figure 1.15), especially if the UI components are created with Interface Builder or XIB file.

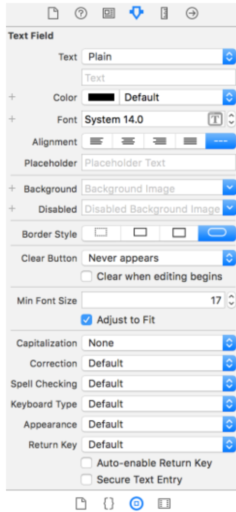


Figure 1.15

This tab helps to make some basic setup on components appearance, view and default behavior etc. In case of creating the custom objects this panel is mostly hidden and can be called back if needed.

4) **Debug** gives information of results and states of various variables during application run.

5) The **Toolbar** holds elements to build, run, test and analyze the application. Here we can also select the simulation device or run on actual one. Using the Xcode simulator we can emulate such events as change of GPS coordinates, shake etc.

4. Git and CocoaPods basics

The control version tools are heavily used during development of iOS applications. The Git is one of the most widely used systems that can ease development process by splitting, merging and visualizing the nonlinear project development history.

As we have seen before the local Git repository can be created

during new project creation. Let's briefly discuss main Git basic commands.

There are two types of Git repository – local and remote. The local repository is a .git catalog and if it was not created with project, we can navigate in console to the project folder and with «git init» make it initialization. If we need to copy information from nonempty parental remote repository - put link on it and use git clone command.

Basically there are three types of objects in git repository – file, tree and commit. File is a version of a user file, tree is a group of user files from different catalogs and commit is a tree with some additional information.

The "git commit" and "git merge" are the most frequently used Git operations:

- "git commit" saves changes to the local repository. The Git requires to explicitly show what exactly must be saved with this commit, thus we need to use "git add" command previously. For example, we need to save changes in HelloWorldVC.swift to the local repository. To do that use following commands, where "-m<message>" presents a comment to the commit:

```
git add HelloWorldView.swift
git commit –m "Create custom view for basic screen"
```

If we need to add changes from many different files, the prefix "-a" can be applied that add all changed files to this commit:

```
git commit –a –m "Create custom view for basic screen"
```

- git merge can be applied to merge parallel tree branches. For example, we need to create a new branch in project version tree, make several changes on it and then merge this new branch with a main one (master branch). To do so, we can use the following commands:

```
# create a branch new
git checkout –b new master
# commit of all changes made in branch new
git commit –a –m "Change and add some
```

```
features"
# merge of branch new with master
git checkout master
git merge new
# deletion of branch new
git branch -d new
```

It should be noted that almost all common Git operations are processed locally and can be synched with remote repository using push and pull commands. The push sends new data from local repository to the remote. It should be noted that remote repository must have only up to date information. If it was changed, first call the git pull command that will load all changes in remote repository to the local and merge those changes into the local. The pull command makes a local copy of changes made in remote repository and if one branch has independent history in local and remote repositories the pull will immediately merge it. The fetch command is also heavily used to work with remote and local repositories, as it present a partial pull. The fetch takes changes from the remote repository and copy them to the local. For more details about git commands please visit the official git site [2].

The CocoaPods is a one of the best dependency manager in Swift and Objective-C projects. The CocoaPods provide developers with over 64 thousand libraries that help to easily scale iOS projects. It is build with Ruby and is installable with the default Ruby available on macOS.

To setup CocoaPods we need to update the packages list with update command and as the list became up to date install pods and setup them (Code sample 1.2).

```
sudo gem update -system
sudo gem install cocoapods
pod setup
```

Code sample 1.2

Then navigate to the project folder and create the Podfile with the command vim Podfile. After this check the project folder for the Podfile presence. Now we can use this file to setup CocoaPods simply using pod'NameOfNeededPod'. For more detailed information and list of available libraries visit [3].

1.1.4 Report requirement and tasks

Practical work tasks:

1. Download Xcode from App Store and install it to the local machine.
2. Create the playground project with custom UIView inside. Add a UILabel with "Hello World" text as a subview to the custom UIView and change it text and background color.
3. Create a HelloWorld swift project as in practical work steps. As you have added changes to it commit them to the master branch. Create a new branch and on this branch add a UILable to the Interface builder. Commit this change and merge the new branch with a master.
4. Install CocoaPods. From the Podfile in HelloWorld project install AFNetworking and SwiftyJSON pods.

The report should contain following sections:

1. Introduction – background, theory and practical work purpose;
2. Development – screenshots with explanation of each practical work task completion, code from ViewController.swift file and screenshot from HelloWorld Interface Builder with simulator screenshot.
3. Summary – conclusions and result summary.

1.1.5 Test questions

1. What is a playground in Xcode? For what purpose it can be applied?
2. What types of template projects does Xcode provide?
3. What is an Interface Builder? How we can change controls common attributes in it?
4. Explain three types of Code Editor presentation. What is the main difference between them?
5. What is Git? Explain git commit, git merge, git push, git pull and git fetch commands.
6. What is a CocoaPods?

1.1.6 Recommended literature and resources

1. Matt Neuburg. Programming iOS 12: Dive Deep Into Views, View Controllers and Frameworks/ o'Reilly Media, 2018 – 1176 p.
2. Git. Documentation. https://git-scm.com/docs
3. CocoaPods. https://cocoapods.org/

Practical work 1.2

DESIGN AND BASIC LAYOUTS OF THE IOS DIABETIC TRACER APPLICATION "GLUCOSE"

1.2.1 Synopsis

This practical work presents a step-by-step analysis of how the health-related application can be designed according to Apple Review Guidelines and official HIG recommendations.

1.2.2 Brief theoretical information

Apple gives a list of requirements and recommendations that have to fulfilled during the development of an application that is planned to be subscribed to App Store [1,3]. Those questions include guidelines arranged into five sections: safety, performance, business, design and legal. As for this and following practical works 1.3 – 1.6 the key application that helps to trace the everyday glucose level in blood we need to focus on section «5.1.3 Health and Health Research» in App Store Review Guidelines.

They provide some special rules to ensure customer privacy is protected:

- apps may not use or disclose to third parties data gathered in the health, fitness, and medical research context—including from the Clinical Health Records API, HealthKit API, Motion and Fitness, MovementDisorderAPIs, or health-related human subject research, etc. The specific health data that is collected from devices should by disclosed.

- apps must not write false or inaccurate data into HealthKit or any other medical research or health management apps;

- personal user health information may not be stored in iCloud;

- apps conducting health-related human subject research must obtain consent from participants or, in the case of minors, their parent or guardian [1].

Accounting those recommendations can remove inappropriate functions which are forbidden by App Store as well as decrease the application development time.

Now, let's set the basic functions that diabetic tracer application «Glucose» should provide:

- Give a information about previously made glucose test in a table form;
- The user should have an option to edit this glucose history table;
- Manually add new glucose measurement: set data in ml/dg, dependence on meal, data and time;
- Synchronize an application with glucometers and get data from those third-party devices;
- Ability to send reminders for the next glucose measurement time;
- Connect to the Health application and get data about latest trainings and heart rate measurement.

The internal data storage will be organized using Core Data store. Within this practical works the «Glucose» application will use mainly native iOS 12 components.

1.2.3 Practical steps

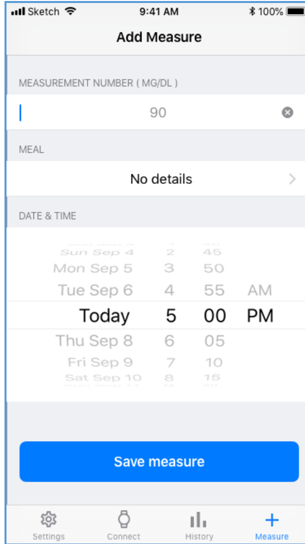
1. Measure page design

Now let's discuss the basic organization of application pages. As we can see from previously made app functions list there are four logical groups of functions – management of the glucose history data, addition of a new measurement, synchronization with devices and few settings. Thus we can present app in as set of four tabs – Settings, Connect, History and Management. The app is heavily dependent on the most frequently used page that allows user to add a new glucose measurement, thus Measure tab will be the first screen that appears after app launch.

The Figure 2.1 presents tab Measure that is basically based on native iOS 12 components. User can add a new measure with UITextField that holds an example of the required data. Selection of meal dependence can be organized with UIAlertController (Figure 2.2) that keep user on the same page and avoid additional forward/backward navigation on page.

Note, that in official Human Interface Guidelines (HIG) [2] the default Cancel button is strictly recommended during the work with Action Sheets component. We can set the meal dependence field as non-required because user can simply forget when this measurement was made, thus Cancel button will switch to the default "No details" variant. The UIDatePicker component is applied to select date and time. This is also an optional field that will keep the default value of date and time

when measurement was saved in the application.



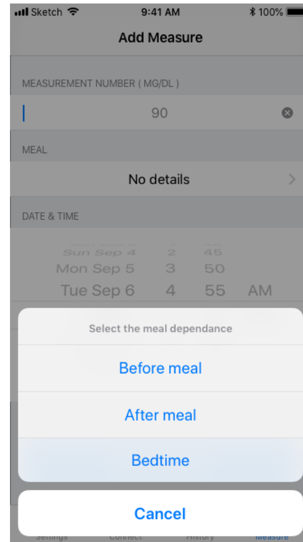


Figure 2.1 Figure 2.2

One of the basic HOG recommendations on Date Picker element use is to decrease the time interval if it is possible, thus we will set the time interval to value 5 on the scale [0; 59] (Figure 2.1).

2. History page design

In the History tab user can view and edit the data on previously made glucose measurements (Figure 2.3). For this purpose, we use UiTableView component. Each cell of the table holds three labels that are based on gathered data. The unnecessary row can be deleted from the table.

3. Connect page design

The user can not only manually add data but also get it from third-party glucometers or other appropriate devices with Bluetooth inside. The initial page presents list of devices that have been already connected. «Add device» button can be used to search for new connection (Figure 2.4).

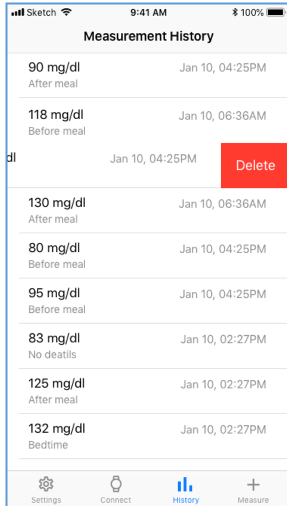


Figure 2.3

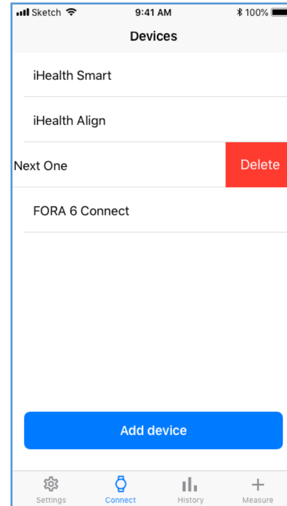


Figure 2.4

While searching the nearby devices via Bluetooth user can view all appropriate for connection devices in a table and by selecting one get connection request (Figure 2.5 – 2.6).

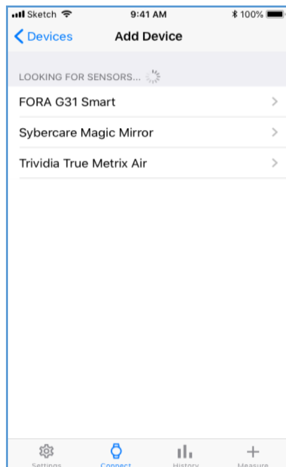


Figure 2.5

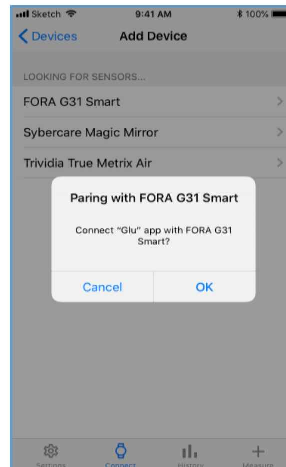


Figure 2.6

The process of search is presented to user with Refresh Connect control, and according to HIG the list of found devices should be constantly updated. The Figure 2.6 shows an Alert message that asks for permission to connect with selected peripheral device. In HIG we can see the recommendation to use mainly two-button Alerts with clear and short names for message and buttons. Additionally, user can dismiss this message by simply moving to another tab.

4. Settings page design

In the Settings tab user can activate native iOS notifications to send reminders on next glucose measurement time (Figure 2.7 – 2.8). For this practical works we will forward user to the Notifications tab in iPhone device settings. There is a list of official recommendations in HIG that can help to increase an effectiveness of notifications instrument [2]. One of the most important conditions, that can be found in this list, are notifications relevance, absence of multiple notifications for the same thing, even if the user hasn't responded and recommendations on badging use. To expand application functionality, we can also ask for authorization to Health data, if Health is available on current user device and collect data from last trainings and heart rate measure.

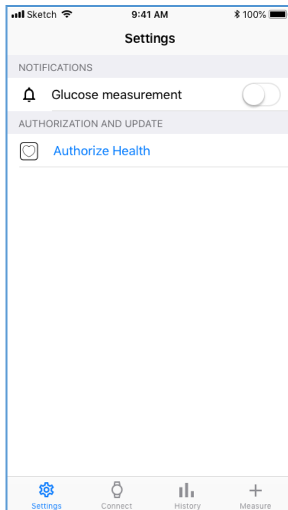


Figure 2.7

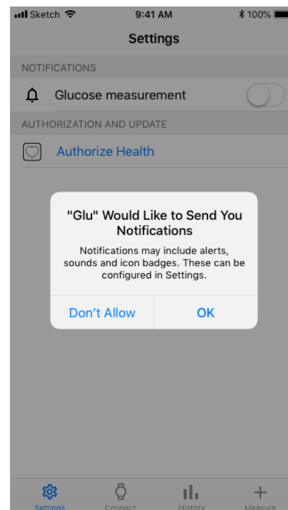


Figure 2.8

1.2.4 Report requirements and tasks

Practical work tasks:

1. Download Sketch or Figma, install the software and design the basic «Glucose» application screens. You can use the partial or full design and data organization of «Glucose» application as it was presented in 1.2.3 Practical steps. Use thenounproject.com and material.io to find icons for buttons and other control elements.
2. Read the official HIG requirements and recommendations for following controls: buttons, labels, pickers, refresh content controls, switchers and text fields. Read the official HIG requirements and recommendations for following views: action sheets, alerts, tables.
3. Add into the Settings tab following additional setup functions: select the glucose units from mg/dL to mmol/L; clear measurements history; delete measurement history for data later than month ago; setup reminder inside an application.

The report should contain following sections:

4. Introduction – background, theory and practical work purpose;
5. Development – screenshots with explanation of each practical work task completion.
6. Summary – conclusions and result summary.

1.2.5 Test questions

1. What is an official Apple HIG and what type of information it presents?
2. What are the basic requirements in Apple Review Guidelines to the health-related applications?
3. What type of information is presented in Measure screen? Why did you used such controls to get user data?
4. What type of information is presented in History screen? How we can alternate the data presentation in this screen?

1.2.6 Recommended literature and resources

1. App Store Review Guidelines. https://developer.apple.com/app-store/review/guidelines/
2. Human Interface Guidelines for iOS https://developer.apple.com/design/human-interface-guidelines/ios/overview/themes/
3. Joshua Greene. Design Patterns by Tutorials: Learning Design Patterns in Swift 4/ Razeware LL, 2018. – 364 p.

Practical work 1.3

TRANSLATING DESIGN INTO CODE - ADD AND SETUP BASIC APPLICATION COMPONENT

1.3.1 Synopsis

In this practical work we will focus on presenting of already created application design inside Xcode Interface Builder. We will setup UI components, set constraints for each view, ink the appropriate views with outlets and actions in code. Finally, we will apply the initially required methods.

1.3.2. Brief theoretical information

After design creation and as we have screens for each application reaction on various user interactions with it, it is time to create a new project file and setup user interface in Xcode. This can be performed through Interface Builder storyboard that gives rather flexible tools to customize and set controls on View Controllers or without storyboard. In the second case we can create and customize all views from code using various imbedded Xcode frameworks or download appropriate from CocoaPods. Xcode doesn't permit usage of both storyboard files with no-storyboard designed views, thus we can easily combine the needed variants. As «Glucose» application consists of four main screens, does not provide user with wide functionality and it was designed upon native iOS 12 element – we can use storyboard to construct application interface.

1.3.2. Practical steps

1. Project creation and Interface Builder overview

Create new single view application project in Xcode. Check the Core Data during new project creation as we will need it later to store user data. Accept the Git repository creation if you will use version control during application development.

Inside of new project go to the Main.storyboard file (Figure 3.1) that presents a default View Controller at the Interface Builder canvas [1].

Interface Builder consists of four main areas:

a) Interface elements hierarchy along with their constraints (placing rules);

b) Interface elements canvas;

c) Toolbar for setting the items constraints;

d) Selection of devices on which the application will run and device orientation modes;

e) Tools to setup elements connections, behaviors, appearance, basic graphical features etc.

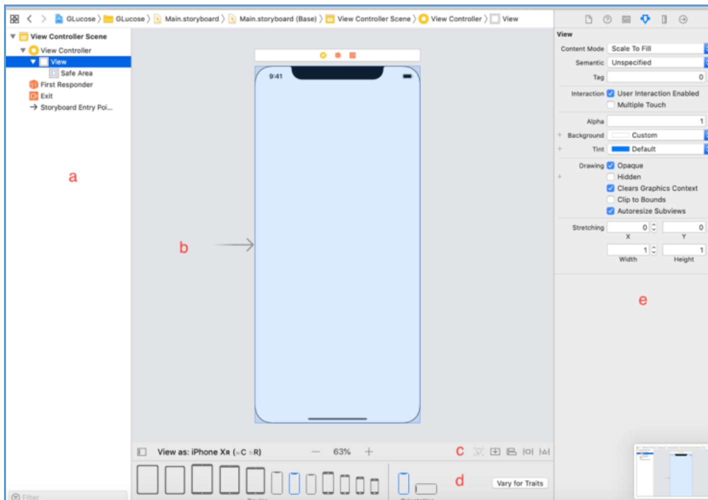


Figure 3.1

An app consist of four tabs, thus we can use the UITabBarController to present it's content. Imbed it into the project with Editor – Embed In – Tab Bar Controller. As a result you can see that Main.storyboard file now presents UITabBarController connected with empty View Controller (Figure 3.2).

Drag from the Objects library two more View Controllers and one Table View Controller for Settings page. Add to each new controller the Tab Bar Item and holding ctrl create segue between Tab Bar Controller and each new controller selecting the«Relationship segue – view controllers» (Figure 3.3).

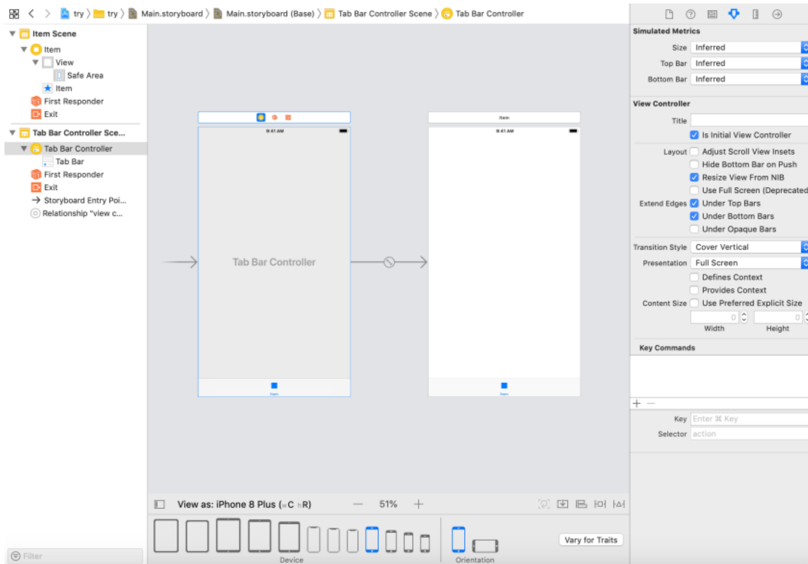


Figure 3.2

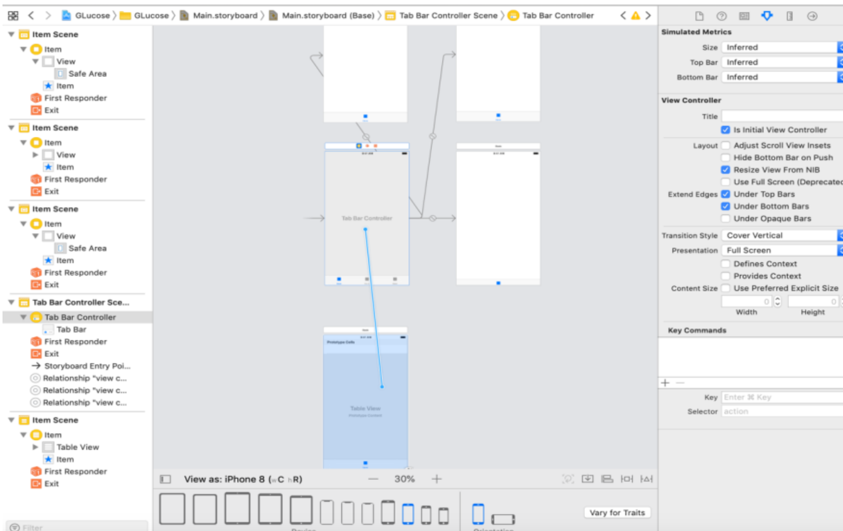


Figure 3.3

2. Constructing the Measure tab in the Interface Builder and programming the basic connections between view and code

First, add to the Assets.xcassets file the app icons in three sized (1x, 2x и 3x) [2] and using Attribute Inspector in the Interface Builder assign icons to each Tab Bar Item or newly created controllers. Change the Title of each.

The tab Measure consists of three fields – measurement number (mg/ml), meal and date and time. This page can be presented as a static table with determined number of sections and rows or as following views hierarchy, namely:

- Navigation bar with “Add Measure title”;
- Top UIView with imbedded UILabel titled as “Measurement number (mg/ml) inside”;
- UITextField with centered text alignment and placeholder “90”;
- Second UIView with UILabel titled as “Meal” inside;
- UIButton that calls for UIAlertController with “No details” title;
- Third UIView with “Date & Time” UILable inside;
- UIDatePicker on mode “Date and time” and 5 minute interval;
- “Save measure” UIButton placed on page footer.

Let’s consider the second presentation type to show the Interface Builder flexibility. Construction and initial setup of UITableView will be considered in the description of the History tab.

3. Constructing the Measure tab in the Interface Builder

Set up UINavigationBar. Select in Objects library NavigationBar and drag it to the Interface Builder canvas. Resize and position it as needed. After you have placed view on canvas Interface Builder will automatically create a set of prototyping constraints that define the view’s current size and position relative to the upper left corner. This can be done for fast prototyping purpose, as by now app can be build and run, but should always be replaced by own explicit constraints. After creation of the first explicit constraint, the system will remove all prototyping constraints from views referred by the constraint.

Interface Builder provides four Auto Layout tools, namely Stack, Align, Pin and Resolve Auto Layout Issues in the bottom-right corner of the Editor window:

1) The Stack tool (Figure 3.4) allows to quickly create a stack view by selecting one or more items in layout and clicking on Stack tool

button. Interface Builder created a stack view from selected items and resizes the stack to its current fitting size based on its contents.

2) The Resolve Auto Layout Issues tool (Figure 3.5) provides a number of options for fixing common Auto Layout issues. The top options affect only the currently selected views. The bottom options will affect on all views in the scene.

3) The Align tool (Figure 3.6) is used to quickly align items in layout. After selection of the items that have to be aligned click Align tool and choose among presented alignment types the appropriate. The Interface builder will create the constraints needed to ensure those alignments.

4) The Pin tool (Figure 3.7) let us quickly define a view's position relative to its neighbors or define its size. Select the item that has to be pinned and call Pin tool from Editor Window. Interface Builder presents a popover view containing a number of options. The top part of popover helps to pin selected item's Leading, Top, Trailing, or Bottom edge to its nearest neighbor. The associated number indicates the current spacing between the items in the canvas. The lower part let us set the items' width and height.

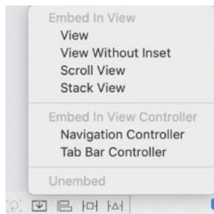


Figure 3.4

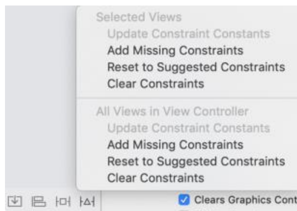


Figure 3.5

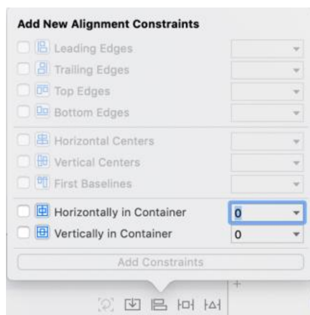


Figure 3.6

Figure 3.7

For UINavigationBar set height and pin it to the Leading, Trailing and Top canvas edges with 0. Change it title to "Add Measure".

Set up top UIView with UILabel "Measurement number (mg/dl)". The same principles are used to set constraints on the rest of views. Select in Objects library the UIView and drag it to the Interface Builder canvas. Set it under the UInavigationBar and give it the proper size. Change the background color, pin it to the bottom of UINavgationBar, Leading and Trailing edges of the screen and give it the proper height. Align it horizontally in container view.

Drag the UILabel from the Objects library and set it inside the UIView by pinning it to the UIViews Bottom and Leading edges. Change title and give it the needed height.

Set up UITextField. Drag the UITextField from the Objects library to the designed screen. Pin it Top to the Bottom of previously added UIView, Leading and Trailing to the edges of container view. Set proper height and align horizontally in container.

As the UITextField has no initial text inside use put "90" to the placeholder settings in the Attribute Inspector. Select None as Border Style, set Font Size and make centered text alignment.

Set up UIView with UILabel "Meal". This UIView and UILabel can be added to the designed screen the same way as UIView with UILabel "Measurement number (mg/dl)". The mane difference is that Top edge of UIView must be pinned to the Bottom edge of UiTextField.

Set up UIButton with title "No details". Drag the UIButton to the Interface Builder canvas. Place it under the last added UIView and pin it Top to the UIView Bottom edge, Leading and Trailing edges to the container view. Set height, change title to "No details" as this parameter goes as default is user will not choose the meal dependence option. The UIButton should be in enables state. Give it the needed background color and title color.

Set up UIView with UILabel "Date & Time". Use the UIView with UILabel "Measurement number (mg/dl)" as an example, with main difference is that Top edge of UIView should be pinned to the Bottom edge of UIButton.

Set up UIDatePicker . Select the UIDatePicker in the Object library and drag it under the last added UiView. Pin it edges as follows: Top edge to the Bottom of UIView, Leading and Trailing edges to the container view and set needed height. Set the horizontal in container

alignment in the Align tool.

Make additional settings in the Attributes Inspector: set 5 minutes interval, the current date must be applied as default; alignment should have center value for both horizontal and vertical lines.

Set up UIButton with title "Save measure". Drag the UIButton from Objects library and place it in the bottom part of the designed screen. Pin it Bottom edge to the bottom of the container view with needed margin, for instance 28, Leading and Trailing edges to the container view with equal margin sizes, for example 16. The UIButton should be horizontally aligned in container. Set it height size, change title to the "Save measure", check the state as it should be enabled, set background color and title color.

4. Connecting the Measure views with code

After placing all views in the Measure tab they should be connected with the code in ViewController [3]. The XCode have already created be default following files:

- AppDelegate.swift. As in IOS the delegate is a class that does something on behalf of another class, and the AppDelegate is a place to handle special UIApplication states, with a lot of functionalities inside.

- ViewController.swift. This file manages the app interface setup and interaction between interface and underlying data.

- Main.storyboars. This file contains the canvas for building and setting up the user interface.

- Assets.xcassets. Holds images that are used in application.

- LaunchScreen.storyboard. This file contains an initial user interface that is loaded when the user taps on app's icon. The system displays launch screen immediately, letting the user know that app is now launching. When app is ready, the system hides the launch screen and reveals app's actual interface.

- Info.plist. This document describes the keys and corresponding values that can be included in an information property list file.

- yourAppName.xcdatamodeld. This file is created by default is during the project creation the use of Core Data was enabled and contains tools for creating and managing database models.

As the MVC is the officially recommended architectural pattern the files can be separated in three folders by its Model, View or Controller features. Each screen (view) presented in the Interface builder

is connected to its own ViewController that will manage the views inside of it. As app consist of four tabs – there have to be four ViewControllers, each managing its own tab. To see this connection go to Main.storyboard, select needed tab and in Identity inspector check the class (ViewController) (Figure 3.8).

The Assistant Editor tool can be applied to ease the connection of views to code. Call the Assistant Editor and place Main.storyboard with designed Measure tab on the left-hand side and connected to Measure tab ViewController on the right-hand side (Figure 3.9).

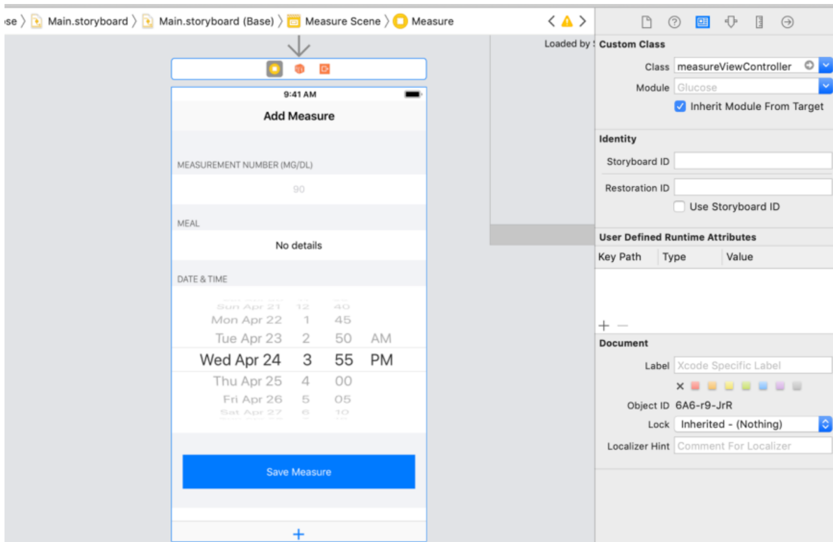


Figure 3.8

There are four controls on Measure tab that give us user information:

- UITextField that present ongoing glucose measure;

- UIButton that calls the UIAllertController for selection of meal dependence;

- UIDatePicker to select the date of ongoing glucose measurement;

- UIButton to save the entered data.

The UITextField and UIDatePicker views are can be refereed in

ViewContoller as IBOutlet. In iOS an outlet is a property of an object that references another object. The reference is archived through Interface Builder. The containing object holds an outlet declared as a property with the type qualifier of IBOutlet and a weak option.

```
@IBOutlet weak var glucoseMeasure: UITextField!
@IBOutlet weak var datePicker: UIDatePicker!
```

Code sample 3.1

The UIButtons are referred in ViewController as IBAction that perform function on its activation.

```
@IBAction func mealBtnPressed(btn: UIButton!){}
@IBAction func saveMeasureBtnPressed(btn:
UIButton!){}
```

Code sample 3.2

After adding IBOutlet and IBActions for each component they have to be connected with according views in Interface Builder. This can be made by holding the dragging from the circle created near the new @IBOutler or @IBAction to the view (Figure 3.9).

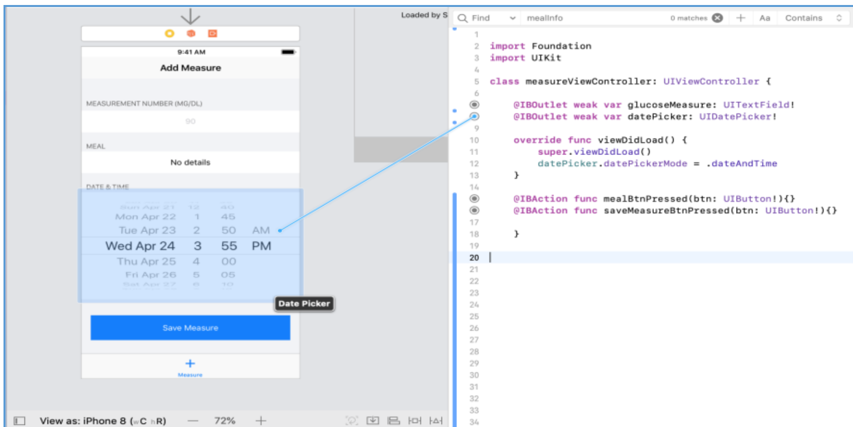


Figure 3.9

5. *Receiving data from Measure views*

As the connection was enabled we can start getting the user information from those controllers. To do so we can create three variables that will hold received data for following use as Core Data entities (Code sample 3.3):

```
var glucoseData: Double?
var meal = "No details"
var date: String?
```

Code sample 3.3

Let's consider the process of obtaining the glucose measurement data from UITextField (Code sample 3.4). The *guard* statement combines two powerful concepts: optional unwrapping and where clauses, thus giving a safe way of avoiding nil, invalid values or the very long *if let* statement.

```
    func getGlucoseMeasure(){
guard let measure = Double(glucoseMeasure.text!)
else {
print("Not a number: \(glucoseMeasure.text!)")
     return
     }
    self.glucoseData = measure }
```

Code sample 3.4

As a result of code sample 3.4 the constant measure of double type is obtained and if user skips this measure we can process what to do next, for instance – give the warning message (Code sample 3.5).

```
func alertMessageMeasurement(){

let alert = UIAlertController(title:
"Oops...", message: "Please enter the
glucose measurement", preferredStyle:
UIAlertController.Style.alert)
```

```
    alert.addAction(UIAlertAction(title: "OK",
    style: UIAlertAction.Style.default,
    handler: nil))

    self.present(alert, animated: true,
    completion: nil)
    }
```

Code sample 3.5

When user presses UIButton to enter the measurement dependence on eaten meal we present UIAlertController [4]. The UIAlertController shows four options: “After meal”, “Before meal”, “Bedtime” or “Cancel” that will chance the UIButton title thus showing user the selected variant. The “Cancel” will leave the default data – “No details”. This can be performed in following way (Code sample 3.6).

```
@IBAction func mealBtnPressed(btn: UIButton!){

   let optionMenu = UIAlertController(title:
nil, message: "Select the meal dependance",
preferredStyle:
UIAlertController.Style.actionSheet)

   let beforeMeal = UIAlertAction(title: "Before
meal", style: .default, handler: {(action) ->
            Void in
   self.meal = "Before meal"
   btn.titleLabel?.text = "Before meal"
        })

    let afterMeal = UIAlertAction(title: "After
meal", style: .default, handler: {(action) ->
            Void in
    self.meal = "After meal"
    btn.titleLabel?.text = "After meal"
        })

    let bedtime = UIAlertAction(title:
"Bedtime", style: .default, handler: {(action) -
```

```
>
    Void in
    self.meal = "Bedtime meal"
    btn.titleLabel?.text = "Bedtime meal"
        })

     let cancel = UIAlertAction(title: "Cancel",
style: .cancel, handler: {(action) ->
        Void in
   self.meal = "No details"
   btn.titleLabel?.text = "No details"
        })

        optionMenu.addAction(beforeMeal)
        optionMenu.addAction(afterMeal)
        optionMenu.addAction(bedtime)
        optionMenu.addAction(cancel)

        self.present(optionMenu, animated: true,
completion: nil)
        }
```

Code sample 3.6

Date and time from UIDatePicker can be obtained with DateFormetter() class (Code sample 3.7) [5]:

```
     func getDate(){
           let dateFormatter = DateFormatter()
           dateFormatter.dateStyle =
  DateFormatter.Style.medium
           dateFormatter.timeStyle =
  DateFormatter.Style.short
           let strDate =
  dateFormatter.string(from: datePicker.date)
           self.date = strDate
           self.datePicker.endEditing(true)
       }
```

Code sample 3.7

6. Constructing the History tab in the Interface Builder and programming the basic connections between view and code

The tab History is presented as UITableView with UITableViewCell inside which present the following user information:

- UILabel with saved glucose measurement;
- UILabel showing the meal dependence;
- UILabel with data and time of corresponding measurement.

The designed cell can be created using default cell settings but for learning purpose we will consider creation of custom UITableViewCell.

7. Constructing the History tab in the Interface Builder

Set up UINavigationBar. As in previous case with Measure tab select in Objects library UINavigationBar and drag it to the Interface Builder canvas. Resize and position it as needed. Pin it's Top to the Top, Leading and Trailing edges to the container view, set height and give name to the Title "Measurement History".

Set up UITableView. Select in Object library the UITable view and drag it under the UINavigationBar. Change the size and pin Leading, Trailing and Bottom edges to the container view and Top to the Bottom edge of UINavigation bar, give it Horizontal alignment.

Set up UTableViewCell. From the Object library drag on the UITableView the UITableViewCell and place it in the top of the table and change it size.

This cell contains three UILabels, thus we need to place them properly and set all constraints. The first UILabel that contains glucose measurement appears in top left corner of the cell, thus pin it Top and Leading edge to the UITableViewCell, give it height and with Aspect ratio make it half-long of cell width.

Place second UILabel under the measures label and pin it Top edge to the Bottom of first one with margin 8, Leading edge to the cell, set height and make as long as previous label.

The last UILabel which presents date and time have to be placed in the upper right corner. Pin it Trailing to the cell, Leading edge to the Trailing of first UILabel with measurement data and make the right text alignment.

Set text colors for each created labels, as well as consider text truncation mode. The result is presented on Figure 3.10.

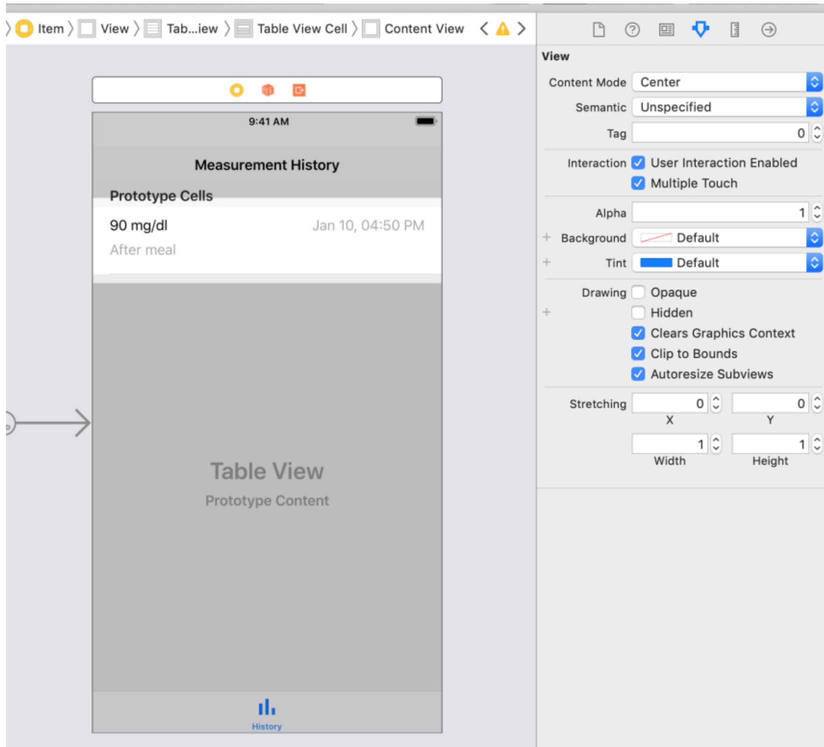


Figure 3.10

8. Connecting the History views with code

After placing the UITableView and labels in UITableViewCell in the History tab they should be connected with the code in ViewController.

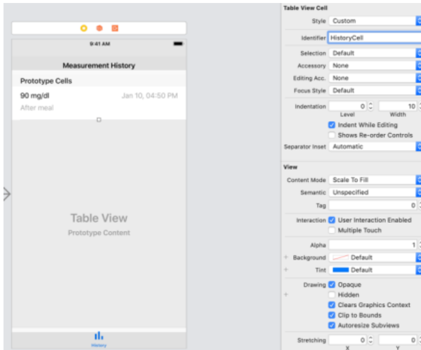
As the table presents cells, let's finish the configuration of the UITableViewCell first. To do so the cell reuse identifier and UITableViewCell subclass, which provide the custom cell behavior, should be created.

Select the created cell in Main.storyboard file and in Attributes inspector define the name of table cell Identifier, for example "HistoryCell" (Figure 3.11).

Create new swift file with File/New/File and add initial connecting information as in Code sample 3.8. Return to the Main.storyboard and

select the UITableViewCell. Choose the Identity inspector and select the file cell View Controller (Figure 3.12).

With the help of Assistant editor create connections from IBOutlets to each UILabel.



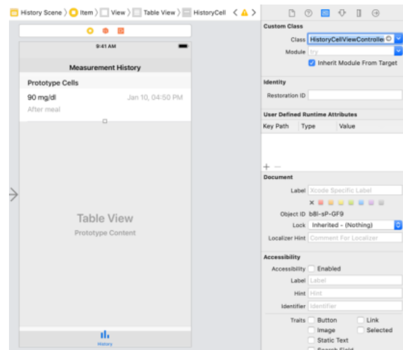


Figure 3.11 Figure 3.12

```
import Foundation
import UIKit

class HistoryCellViewController:
UITableViewCell {

    @IBOutlet weak var measure: UILabel!
    @IBOutlet weak var meal: UILabel!
    @IBOutlet weak var date: UILabel!

    override func awakeFromNib() {
        super.awakeFromNib()
    }
}
```

Code sample 3.8.

Now let's consider the initial configuration of UITableView [6]. Tables are data-driven elements of an interface. We need to provide app with data, along with the views needed to render each piece of that data onscreen, using a data source object that adopts the required UITableViewDataSource

protocol. The table view arranges views onscreen and works with data source object to keep that data up to date.

The second required protocol to work with UITableView is UITableViewDelegate. The following features can be managed while accessing methods from this protocol:

- Creation and managing custom header and footer views.
- Specifying custom heights for rows, headers, and footers.
- Providing height estimates for better scrolling support.
- Indent row content.
- Respond to row selections.
- Respond to swipes and other actions in table rows.
- Support editing the table's content.

The table specifies rows and sections using NSIndexPath objects.

When we specify usage of UITableViewDataSource there are two methods are required (Code sample 3.9):

```
    func tableView (_ tableView: UITableView,
numberOfRowsInSection section: Int) -> Int
    func tableView (_ tableView: UITableView,
cellForRowAt indexPath: IndexPath) ->
UITableViewCell
```

Code sample 3.9

After providing the basic UITableView setup to the History tab View Controller the file will contain following changes (Code sample 3.10):

```
    import UIKit
    import CoreData

    class HistoryViewController:
UIViewController, UITableViewDataSource,
UITableViewDelegate {

    @IBOutlet weak var tableView: UITableView!

        override func viewDidLoad() {
            super.viewDidLoad()
            tableView.delegate = self
```

```
            tableView.dataSource = self
        }
override func viewDidAppear(_ animated: Bool) {
        tableView.reloadData()
        }

        func tableView(_ tableView: UITableView,
cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
    if let cell =
tableView.dequeueReusableCellWithIdentifier("His
toryCell") as? HistoryCellViewController {
            return cell
                }
           else {
                return
HistoryCellViewController()
                }
           }
    func tableView(tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
            return 1
        }
    }
```

Code sample 3.10

1.3.4 Report requirements and tasks

Practical work tasks:

1. Create the new project in Xcode and create the application screens structure using Tab Bar Controller.
2. Set views in Measure tab; apply all necessary constraints so the views will properly operate on various devices. Connect controls with code and set initial methods. Build and run application.
3. Set views in History tab, create constrains for each view and connect them with code. Add methods required by UITableViewDataSource protocol. Build and run application on several iPhone simulators.

The report should contain following sections:

7. Introduction – background, theory and practical work purpose;

8. Development – screenshots with explanation of each practical work task completion. Screenshots of how applications views behave on several iPhone simulators.

9. Summary – conclusions and result summary.

1.3.5 Test questions

1. What is an Xcode Interface Builder? Describe it logical parts.
2. How constraints to different views can be added inside of an Interface Builder? Describe the constraints types.
3. Explain the difference between dynamic and static UITableView?
4. Describe how we can create the segue between different View Controllers.
5. What type of data we can set for UIView, UiButton, UILabel and UITableView from Attributes Inspector?

1.3.6 Recommended literature and resources

1. Using Interface Builder. https://developer.apple.com/library/archive/documentation/ToolsLanguages/Conceptual/Xcode_Overview/UsingInterfaceBuilder/
2. Adding Assets. https://developer.apple.com/library/archive/documentation/ToolsLanguages/Conceptual/Xcode_Overview/AddingImages/
3. Connecting Objects to Code. https://developer.apple.com/library/archive/documentation/ToolsLanguages/Conceptual/Xcode_Overview/ConnectingObjectstoCode/
4. UIAlertController Class Documentation. https://developer.apple.com/documentation/uikit/uialertcontroller/
5. UIDatePicker Class Documentation. https://developer.apple.com/ documentation/uikit/uidatepicker/
6. UiTableView Class Documentation. https://developer.apple.com/ documentation/uikit/uitableview

Practical work 1.4
GETTING STARTED WITH CORE DATA

1.4.1 Synopsis

In this practical work we will discuss what a Core Data is and how we can use in inside of application. We will create the first Entity, add its attributes, save glucose measures to it learn how to retrieve them and present inside of the table.

1.4.2 Brief theoretical information

The Core Data is heavily used in many nowadays mobile applications to save user permanent data for offline use, to cache temporary data etc. The data types and relationships can be defined through Core Data's Data Model editor. This model editor also helps with generation of respective class definitions [1]. Core Data abstracts the details of mapping app's objects to a store, making it easy to save data from Swift and Objective-C without administering a database directly.

As it was stated before, the glucose management app should support deleting the unnecessary data from history table. The Core Data's undo manager can help to track changes and can also roll them back individually, in groups, or all at once, making it easy to add undo and redo support to the app [2].

1.4.3 Practical steps

1. Creation of Core Data Model

The Core Data model was already made during project creation, but if it is not the model can be added to the project in following way:

1. Choose File > New > File and select from the iOS templates. Scroll down to the Core Data section, and choose Data Model (Figure 4.1).
2. Click Next. Name model file, and select its group and targets (Figure 4.2).
3. An .xcdatamodeld file with the specified name is now added to the project (Figure 4.3).

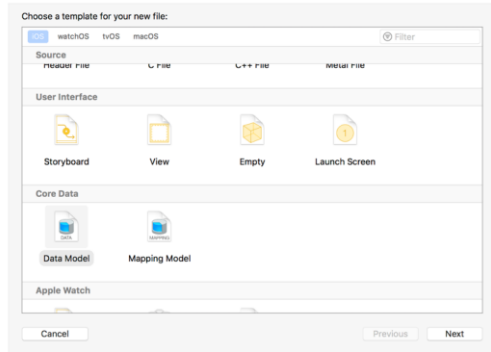


Figure 4.1

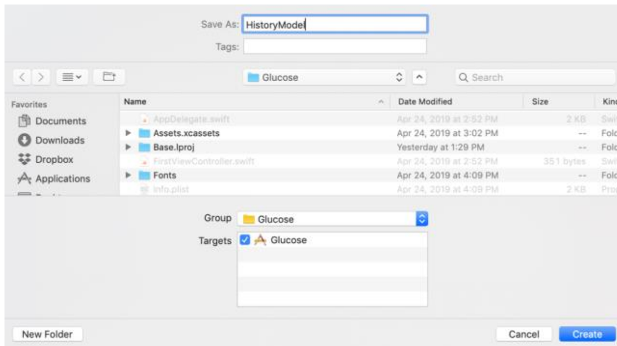


Figure 4.2

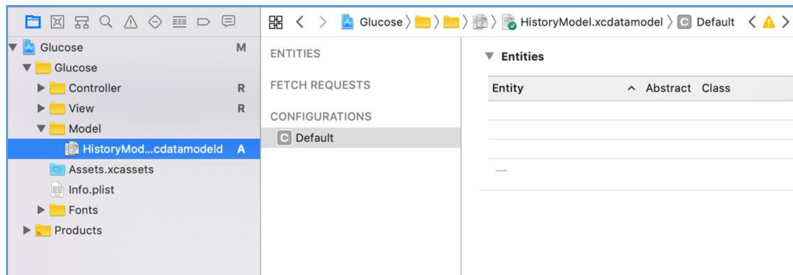


Figure 4.3

To start Core Data Model configuration we need to create its Entity. An entity describes an object, including its name, attributes, and relationships. It should be created for each application object.

Click Add Entity at the bottom of the editor area. A new entity with placeholder name Entity appears in the Entities list. Double-click the newly added entity, and change it name. This updates both the entity name and class name visible in the Data Model inspector. In addition to the required name and class name fields, entities have a default setting for the required code generation field. If inheritance, unique constraints, versioning or other optional information have to be added we need to configure entity attributes.

2. Creation of Entity Attributes

The attributes can be created in following way:

1. Select created entity and with button Add attribute placed in the bottom of editor window add new one.

2. A new attribute with placeholder name attribute, of type Undefined, appears in the Attributes list. In the Attributes list, double-click the newly added attribute, and name it in place.

3. In the Attributes list, as shown in Figure 4.4, click on Undefined and select the attribute's data type from the Type dropdown list.

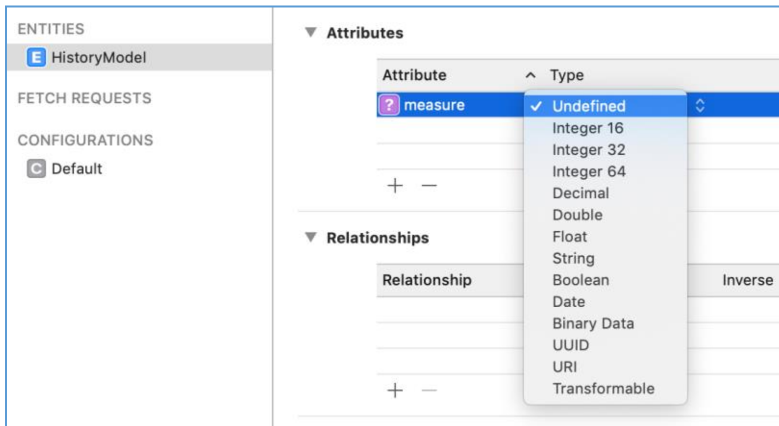


Figure 4.4

We can use the Data Model Inspector (View/Inspectors/Show Data Model Inspector) to configure attributes (Figure 4.5).

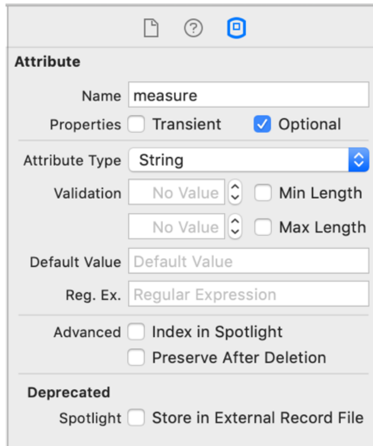


Figure 4.5

By default, attributes are saved to the store. Selecting the Transient attribute property forbids the saving to the persistent store. Transient attributes are a useful place to temporarily store calculated or derived values. Core Data does track changes to transient property values for undo purposes.

Optional attributes are not the same as Swift optionals. Optional attributes aren't required to have a value when saved to the persistent store.

The attribute's data type reflects the selection made in the Attributes list's Type dropdown.

We can optionally set validation rules such as the minimum and maximum values for a numeric type.

Most value types supply a default value. New object instances set the attribute to this default value on initialization, unless another value have already been specified.

In Advanced section with "Index in Spotlight" addition of the field to the Spotlight index for instances created from this entity can be specified. The second option here is "Preserve After Deletion", which includes the attribute in this entity's tombstone.

4. Add two more attributes to the HistoryModel entity that reflect the meal dependence and date with time information.

3. Saving user data to Core Data Model with Save Measure button

Now we can save the data to created HistoryModel entity after user presses Save Measure button on Measure tab. Start with importing CoreData than apply the following changes (Code sample 4.1):

```
@IBAction func saveMeasureBtnPressed(btn:
UIButton!){

        getGlucoseMeasure()
        getDate()

       let app = UIApplication.shared.delegate as!
AppDelegate
      let context = app.managedObjectContext
      let entity =
NSEntityDescription.entity(forEntityName:
"HistoryModel", in: context)!
      let history = HistoryModel(entity: entity,
insertInto: context)
      history.measure = self.glucoseMeasure.text
      history.meal = self.meal.text
      history.date = self.date.text
      context.insertObject(history)

          do {
               try context.save()
          } catch {
               print("Could not save recipe")
          }
self.navigationController?.popViewController(ani
mated: true)
        }
```

Code sample 4.1

4. Configuration of the HistoryCellViewController class

Now we need to finish configuration of HistoryCellViewController class to display the saved information. To do so we need to add following function that will connect UILabel with info from HistoryModel and import CoreData (Code sample 4.2).

```
func configureCell(history: HistoryModel) {
    measure.text = history.measure
    meal.text = history.meal
    date.text = history.date
}
```

Code sample 4.2

5. Configuration of the HistoryViewController class

Create an array with data from HistoryModel first and than function that will fetch and set the obtained from data model results. Call the fetchAndSetResults() with method that reloads table from viewDidAppear() (Code sample 4.3)

```
var history = [HistoryModel]()
    override func viewDidAppear(animated:
Bool) {
        fetchAndSetResults()
        tableView.reloadData()
    }
func fetchAndSetResults() {
let app = UIApplication.shared.delegate as!
AppDelegate
 let context = app.managedObjectContext
 let fetchRequest  =
NSFetchRequest(entityName: "HistoryModel")
 do {
        let results = try
context.executeFetchRequest(fetchRequest)
self.history = results as! [HistoryModel]
    } catch let err as NSError {
            print(err.debugDescription)
        }
    }
```

Code sample 4.3

Change the functions that set cells with data inside the table and show all saved data into the rows (Code sample 4.4).

```
func tableView(_ tableView: UITableView,
```

```
cellForRowAt indexPath: IndexPath) ->
UITableViewCell {

     if let cell =
tableView.dequeueReusableCellWithIdentifier("His
toryCell") as? HistoryCellViewController {

      let history = history[indexPath.row]
      cell.configureCell(history)
       return cell
                    }
            else {
                  return
     HistoryCellViewController()
                    }
           }
     func tableView(tableView: UITableView,
numberOfRowsInSection section: Int) -> Int {
             return history.count
          }
```

Code sample 4.4

1.4.4 Report requirements and tasks

Practical work tasks:

1. Check if a Core Data model was created along with project. If it is not, add Core Data to application as it was discussed in section 1.4.3.
2. Create an Entity with attributes that will present a glucose measurement that was added by user to an application.
3. Add methods to save the data inside Core Data model and retrieve it to fill the table on History tab with data.
4. Add methods to delete the selected by user data from Core Data.

The report should contain following sections:

1. Introduction – background, theory and practical work purpose;
2. Development – screenshots with explanation of each practical work task completion; MeasureViewController.swift and

HistoryViewController.swift code with comments; screenshot of working application on several iPhone simulators.

3. Summary – conclusions and result summary.

1.4.5 Test questions

1. What is a Core Data?
2. Explain the idea of Core Data entity and its attributes. How a new entity can be created in swift project?
3. What methods are applied to make a new data record to the Core Data model?
4. How we can read data from Core Data?
5. What method can be applied to delete specific data from Core Data model?

1.4.6 Recommended literature and resources

1. J.D.Gauchat. Core Data in iOS 12/ MinkBooks, 2018. – 60 p.
2. Core Data. Framework Documentation. https://developer.apple.com/documentation/coredata

Practical work 1.5

ACCESSING USER HEALTH INFORMATION USING HEALTHKIT

1.5.1 Synopsis

In this practical work we will focus on interaction with HealthKit framework, specifically how an application can access it, query for its data samples and save the results.

1.5.2 Brief theoretical information

The glucose level depends not only on nutrition, but on physical activity as well, thus designed management app needs an access to the the data stored in HealthKit.

HealthKit provides a central repository for health and fitness data on iPhone and Apple Watch. The applications can communicate with HealthKit data only with user permission to access and share its data. The framework was designed basically to share data between apps so it contains the types of data and units to a predefined list, thus developers cannot create custom data types or units using only those types that HealthKit provides.

The framework uses a large number of subclasses, which produces deep hierarchy of similar classes with small but meaningful differences between them. There are also closely related classes in HealthKit that must be paired correctly.

The HealthKit saves a variety of data types to the HealthKit Store:

- *Characteristic data* – presents items that are constant, such as the birthdate, blood type, biological sex, and skin type. This data can be accessed directly from HealtHkit store using such methods as the dateOfBirth(), bloodType(), biologicalSex() and fitzpatrickSkinType(). The application cannot save this data type as the user must enter or modify it with Health app directly. Your application cannot save characteristic data.
- *Sample data* – most users' health data is stored in samples that represent information at some particular moment of time. All sample classes are subclasses of the HKSample class, which is a subclass of the HKObject class.
- *Workout data* – data on fitness activities is stored as HKWorkout samles, which is also a subclass of HKSample.

- *Source data* – every sample stores data about its source. The HKSourceRevision object contains info about each app or device that saved those samples and the HKDevice object contains info about the hardware device that produced the data.
- *Deleted objects* – the HKDeletedObject is used to temporarily store the UUID (the unique identifier for some particular entity) of an item that was deleted from the HealthKit store.

The HKObject class is the superclass of all HealthKit sample types and all HKObject subclasses are immutable.

Each object of this class has the following properties:

- *UUID* – unique identifier for the particular entry.
- *Metadata* – dictionary that contains additional information about the entry.
- *Source Revision* – the source (device that directly saves data into HealthKit or application) of the sample.
- *Device* – the device that creates the data stored in the sample.

The HKSample class is a subclass of HKObject. Sample objects present data at a some point in time, and all sample objects are subclasses of the HKSample class, with following properties:

- *Type* - the sample type, such as a sleep analysis sample or a step count sample.
- *Start date* - the sample's start time.
- *End date* - the sample's end time. If the sample represents a single point in time, the end time should equal the start time.

To use HealthKit in an application is have to be enabled, checked it is available on current device, the app's HealthKit store should be created and an app must send a request for permission to read and share data [1].

1.5.3 Practical steps

1. *Enable HealthKit*

To start using the HealthKit, we need to add HealthKit capabilities for your app. In Xcode, select the project and turn on the HealthKit capability (Figure 5.1). The Health Records checkbox must be enabled only if an app needs to access the user's clinical records. It should be noted that during App Review application can be rejected is Health Records were enabled but app actually doesn't uses the Health Record data.

After enabling HealthKit in application, Xcode will add HealthKit to the list of required device capabilities that prevents users from purchasing or

installing the app on devices that do not support HealthKit. In case if HealthKit is not required for the application correct operation we can delete the record *healthkit* from the *Required device capabilities* array in *Info.plist*. After enabling this feature in application we need to check the availability on current device.

In glucose management application user can connect Health through Settings tab, thus we need to build it and make some additional configuration first.

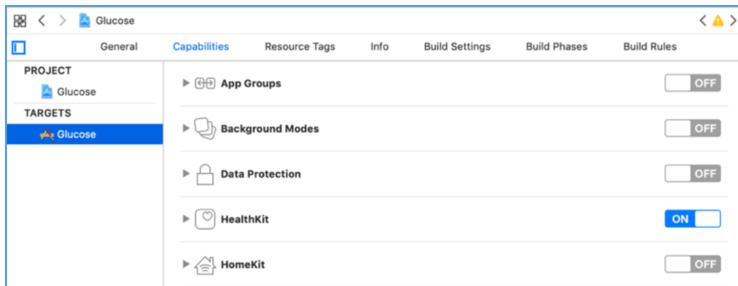


Figure 5.1

2. Constructing Settings tab in the Interface Builder

Set up UINavigationBar. Drag from Object library the UINavigation bar and set it in the top of Settings tab. Pin it Leading, Trailing and Top edges to the container view. Set height and title “Settings”.

Set up the static UITableView. The list of parameters presented on Settings tab is presented in table form that have known number of rows and sections, thus we need to create the static UITableView. The static table views can only be created from UITableViewController that was already added in Practical work 1.3.

Start with dragging UITableView to the Interface Builder canvas. Place and size the table, pin Leading, Trailing and Bottom edges to the container view and Top edge to the Bottom of UINavigationBar.

In the Attributes Inspector of UITableView change content type to Static cells, set 2 sections, and by selecting each section change it name and number of rows. Change the cells height.

Place UIImageView in the top cell. Pin it Leading edge to the container view Leading edge. Set height and width and align vertically in

container. Select image and change the UiImageView content mode to Aspect Fit.

Add UILable near UiImageView. Pin it Leading edge to the Trailing edge of the image with needed margin. Set height and width and align vertically in container. Change UILable title.

Select Switch in the Objects library and place it near UILable. Pin it Trailing edge to the container view, set height, width and align vertically in container view. The Switch should be disabled by default, thus change it state to Off.

Repeat the same with UIImageView in the second section. The Health authorization is performed through UIButton object, thus drug it fron Object library and place near image. Pin it leading edge to the UIImageView Trailing side, set height and width, give vertical alignment in the container view, which is a static cell. Change title. The result should look as on Figure 5.2.

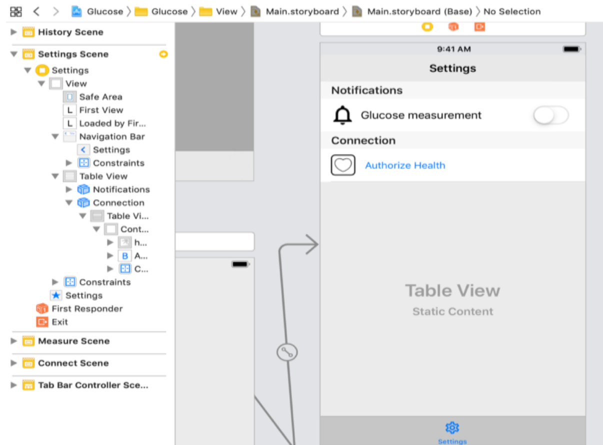


Figure 5.2

3. Connecting the History views with code

To handle events from UISwitch and UIButton we need to create IBOutlets for switch and IBAction for button in code and connect outlet and action with appropriate controls using control-drag from views to the SettingsViewController.swift (Figure 5.3).

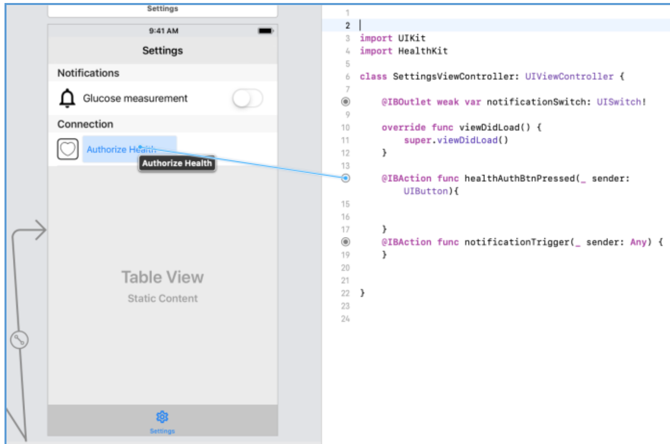


Figure 5.3

4. Ensure HealthKit's availability, create the HealthKit Store and request Permission to read and share data

First we need to create an empty class HealthKitSetupAssistant with an error type and the body of a method that will be used to authorize HealthKit – autgorizeHealthKit(completion:) [2]. The method accepts no parameters and has a completion handler which returns a Boolean value and an optional error. Create new swift file and pass following code inside (Code sample 5.1).

```
import HealthKit

class HealthKitSetupAssistant {
  private enum HealthkitSetupError: Error {
    case notAvailableOnDevice
    case dataTypeNotAvailable
  }
  class func authorizeHealthKit(completion:
@escaping (Bool, Error?) -> Swift.Void) {

  }}
```

Code sample 5.1

To ensure HealthKit availability the isHealthDataAvailable() method should be called in authorizeHealthKit() method (Code sample 5.2). This method should be called before any other HealthKit method as if it is not available, for example iPad does not support the HealthKit, the other methods will fail with “errorHealthDateUnavailable”. If use of HealthKit is restricted on user device HealthKit methods will fail with “errorHealthDataRestricted”.

```
guard HKHealthStore.isHealthDataAvailable() else
{
        completion(false,
HealthkitSetupError.notAvailableOnDevice)
        return
    }
```

Code sample 5.2

HealthKit requires fine-grained authorization to protect the user’s privacy, thus application must request permission to both read and share each data type before you any attempt to access or save the data [3].

For example, in the glucose application we can ask for permission to read and share heart rate, cycling distance, walking or running distance and swimming samples. In order to create an HKObjectType for given biological characteristics or quantity we need to use HKObjectType.characteristics(forIdentifier:) or HKObjectType.QuantityType(forIdentifier:) in authorizeHealthKit() method (Code sample 5.3).

```
guard  let distanceCycling =
HKObjectType.quantityType(forIdentifier:
HKQuantityTypeIdentifier.distanceCycling),
            let distanceWalkingRunning =
HKObjectType.quantityType(forIdentifier:
HKQuantityTypeIdentifier.distanceWalkingRunning)
,           let heartRate =
HKObjectType.quantityType(forIdentifier:
HKQuantityTypeIdentifier.heartRate),
            let distanceSwimming =
HKObjectType.quantityType(forIdentifier:
```

```
HKQuantityTypeIdentifier.distanceSwimming) else
{
completion(false,
HealthkitSetupError.dataTypeNotAvailable)
                return
        }
```

Code sample 5.3

HealthKit expects a set of HKSampleType objects that represent the kinds of data which user can write. The immutable data that can be only read can be presented with HKObjectType objects. Add following code to authorizeHealthKit() method (Code sample 5.4).

```
let healthKitTypes: Set<HKSampleType> =
[distanceCycling, distanceWalkingRunning,
heartRate, distanceSwimming]
```

Code sample 5.4

Now we need to request authorization from HealthKit and then call completion handler from authorizeHealthKit() method (Code sample 5.5).

```
HKHealthStore().requestAuthorization(toShare
: healthKitTypes,
         read: healthKitTypes) { (success,
error) in  completion(success, error)
}
```

Code sample 5.5

As is was stated before the authorizeHealthKit() have to be invoked on pressing the Health authorization button. In SettingsViewController we can add the following code that will print a message to the console to let us know if HealthKit was successfully authorized and updates the button state (Code sample 5.6). The HealthKit cannot be unauthorized directly from the application as it can be disconnected only if user delete app from device or turn off the connection if device Settings.

```
@IBAction func healthAuthBtnPressed(_ sender:
UIButton){

HealthKitSetupAssistant.authorizeHealthKit {
(authorized, error) in

            guard authorized else {

  let baseMessage = "HealthKit Authorization
Failed"

            if let error = error {
 print("\(baseMessage). Reason:
\(error.localizedDescription)")
       } else {print(baseMessage)
                }
                return
            }
print("HealthKit Successfully Authorized.")
            sender.isEnabled = false
           }
         }
```

Code sample 5.6

5. *Querying Samples*

After passing the authorization stage we need to query for most recent samples – heart rate, cycling, walking, running and swimming distances.

Querying the samples from HealthKit splits into two stages:

1. To specify the type of sample you want to query;
2. Set additional parameters to help filter and sort the data.

There are few similarities with Core Data, for example HKSampleQuery is very similar to NSFetchRequest for an entity type.

Once the query is setup we call HKHealthStore's executeQuery() method to fetch the results.

For querying purpose, we will create a single generic function that loads the most recent samples of any type. Create a ProfileDataStore empty class and import HealthKit framework inside. This class represent a point of access to all of the health-related data from HealthKit. Add a getMostrecentSample() method (Code sample 5.7) inside that takes a

sample type, builds a query to get the most recent data of that type. The code in the completion handler occurs inside of a Dispatch block because querying sample from HealthKit is an asynchronous process. We want the completion handler to happen on the main thread, so the user interface can respond to it in other case the application will crash. If all goes well, the query will execute and return a sample to the main thread where SettingsViewController can take that content.

```
import Foundation
import HealthKit

class ProfileDataStor{
    class func getMostRecentSample(for
sampleType: HKSampleType,
   completion: @escaping (HKQuantitySample?,
Error?) -> Swift.Void) {
        let mostRecentPredicate =
HKQuery.predicateForSamples(
           withStart: Date.distantPast,
           end: Date(),
           options: .strictEndDate)
        let sortDescriptor = NSSortDescriptor(
           key: HKSampleSortIdentifierStartDate,
           ascending: false)
        let limit = 1
        let sampleQuery = HKSampleQuery(
           sampleType: sampleType,
           predicate: mostRecentPredicate,
           limit: limit,
           sortDescriptors: [sortDescriptor]) {
           (query, samples, error) in
           DispatchQueue.main.async {

           guard let samples = samples,
                 let mostRecentSample =
samples.first as? HKQuantitySample else
{completion(nil, error)
              return
```

```
                }
                    completion(mostRecentSample,
nil)
               }}
          HKHealthStore().execute(sampleQuery)
        }
      }
```

Code sample 5.7

Now we can locate the displayMostRecentHeartRate() method in SettingsViewController.swift. The method starts by creating a Heart Rate sample type, then pass the sample type to getMostRecentSample() of ProfileDataStore class which returns the latest record from HealthKit (Code sample 5.8). This record can be used for all appropriated purposes inside of an application.

```
func loadMostRecentHeartRate(){
        guard let heartRate =
HKSampleType.quantityType(forIdentifier:
.heartRate) else {
            print("Heart Rate Sample Type is no
longer available in HealthKit")
            return
        }
ProfileDataStore.getMostRecentSample(for:
heartRate) { (sample, error) in
            guard let sample = sample else {
                if let error = error {
            self.displayAlert(for: error)
                }
                return
            }
              }
```

Code sample 5.8

In case if something goes wrong user receives an alert message (Code sample 5.9).

```
    private func displayAlert(for error: Error)
```

```
{
        let alert = UIAlertController(
     title: nil, message:
error.localizedDescription,
               preferredStyle: .alert)
     alert.addAction(UIAlertAction(title: "OK.",
     style: .default, handler: nil))
     present(alert, animated: true, completion:
nil) }
```

Code sample 5.9

1.5.4 Report requirements and tasks

Practical work tasks:

1. Enable the Health Kit in project Settings, check its availability on user device.
2. Create a Settings tab with static UITableView and connect it with outlets and actions in code.
3. When user selects the authorization to Health request an access to the Health Kit data and disable "Authorize Health" button.
4. Download samples for walking and running, cycling and swimming distances. Convert the received data and print it to the console.
5. Create a new Core Data entity that will hold the latest HealthKit samples for walking and running, cycling, swimming distances and a heart rate sample. Save samples measures into created Core Data entity.
6. Add the send notifications feature through triggering UISwitch state change in Setting tab.
7. Advanced task: In navigation bar of History tab add right bar button that will load a new View Controller as shown on Figure 5.4 – 5.5. This View Controller should present received from HealthKit latest data (heart rate, walking and running distance, swimming and cycling distances).

The report should contain following sections:

1. Introduction – background, theory and practical work purpose;
2. Development – screenshots with explanation of each practical work task completion; code with comments from SettingsViewController.swift, MeasureViewController.swift and HistoryViewController.swift files; screenshot of working application on several iPhone simulators.
3. Summary – conclusions and result summary.

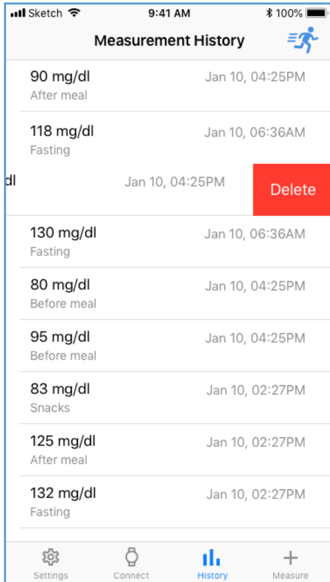


Figure 5.4

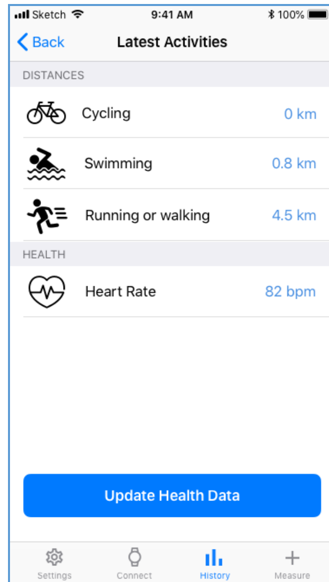


Figure 5.5

1.5.5 Test questions

1. What is a HealthKit framework?
2. What steps should be performed to authorize in HealthKit?
3. How to make basic setup of static UITableView?
4. How we can trigger different the UISwitcher stages?

1.5.6 Recommended literature and recourses

1. About the HealthKit Framework. https://developer.apple.com/documentation/healthkit/about_the_healthkit_framework

2. Setting Up HealthKit. https://developer.apple.com/documentation/healthkit/ about_the_healthkit_framework

3. Protecting User Privacy. https://developer.apple.com/documentation/healthkit/protecting_user_privacy

Practical work 1.6
INTEGRATING THIRD-PARTY DEVICES THROUGH BLUETOOTH

1.6.1 Synopsis

In this practical work you will learn the key concepts of the Core Bluetooth framework to discover, connect and retrieve data from compatible devices such as glucometers or other third-party health trackers.

1.6.2. Brief theoretical information

Connection to the real-world devices such as glucometers, workout equipment, heart-rate monitors can help to gather more accurate information thus provide user with deep insights on his data. While we have already created the way how to manually add data about last glucose measurement into an application, we need to automate it as well. Apple has introduced the Core Bluetooth framework, which can communicate with various third-party devices via BLE (Bluetooth Low Energy) wireless technology [1].

We will use the iHealth Gluco the wireless smart gluco-monitoring system for this laboratory work, but any Bluetooth glucometer should work as well.

A Bluetooth device can be either central or peripheral. The central device receives the data and the peripheral – publishes data that can be consumed by other devices. For this practical work the iPhone 8 with iOS 12.1 will be the central device that receives glucose measurement data from the peripheral.

In form of advertising packets the Bluetooth peripherals broadcast some of the data. These packets basically contain information such as the peripheral's name and main functionality, sometimes providing additional info about the kind of data they can give. The central device scan for these packets, identify any peripherals it finds relevant and connect to individual for more information.

The advertising packets are small thus presenting limited amount of information. To share more data, a central must connect to a peripheral. The peripheral's data splits into to types – services and characteristics, that are represented by UUID that can be 16-bit or 128-bit value:

- *Service* is a data collection and associated behaviors describing a specific function or feature of a peripheral. For example, a glucometer has a Glucose service. Note that peripheral can have more than one service.

- *Characteristics* provide further details about a peripheral’s service. For instance, the Glucose service has a Glucose Measurement characteristic that contains the mg/dl data. Each service of peripheral can have more than one characteristic [2 – 3].

1.6.3. Practical steps

1. *Set up the real device for build and run from XCode*

The iOS simulator doesn’t support Bluetooth, thus we need to build and run on an actual device. First connect the iPhone to MacOS machine with USB line. In XCode go Product/Destination menu item in the top menu bar (Figure 6.1 – 6.2) and then select the real iPhone device under Device menu.

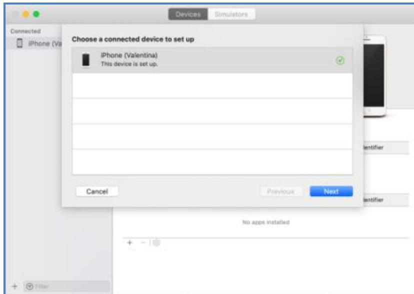


Figure 6.1

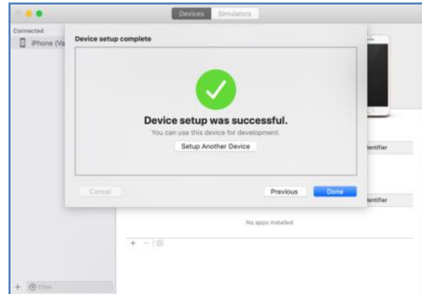


Figure 6.2

2. *Create the Connect tab in the Interface Builder*

The Connect tab contains UINavigationBar, UIView with UILable and Activity Indicator, UITableView with UITableViewCell inside.

Drag the UINavigationBar to the Connect View Controller canvas and place it in the top. Pin it Leading, Trailing and Top to the container view and set height.

Add UIView under the bar. Pin it Leading and Trailing edges to the container view and Top edge to the Bottom of UINavigationBar. Set height and change background color.

Place the UILable inside and pin it Leading and Bottom edges to the container view while keeping the margin. Change title text using Attributes Inspector, set text color. Add Activity Indicator near UILable. Pin it Leading

edge to the label Trailing and Bottom edge to the container view. Set height and width.

Add the UITableView and pin it Leading, Trailing and Bottom to the container view, while Top edge should be pinned to the Bottom of UIView. Drag from Objects library the prototype UITableViewCell inside the table. This cell contains only one UILable inside and while being of custom style uses a disclosure indicator as accessory type. The result is presented on Figure 6.3

Add a new swift file that will present data from UITableViewCell with following code inside (Code sample 6.1). Select the prototype cell in Interface Builder and navigate to Identity Inspector to define a new class to the cell.

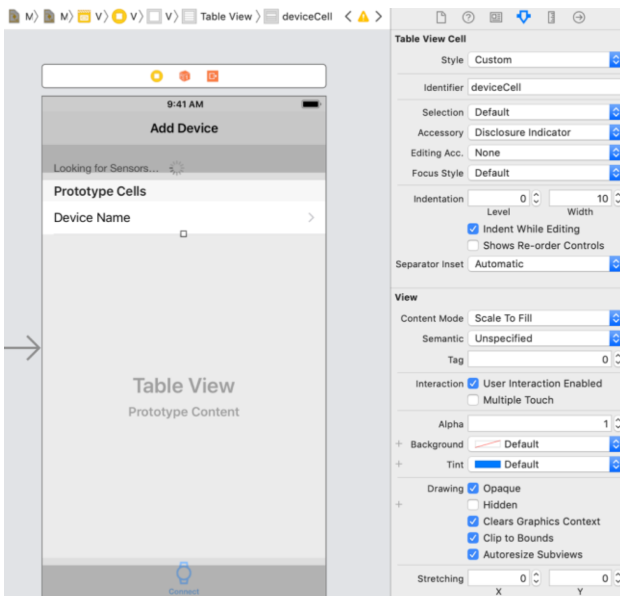


Figure 6.3

```
import Foundation
import UIKit

class DeviceCell: UITableViewCell{
        @IBOutlet weak var nameLabel: UILabel!
```

```
    override func awakeFromNib() {
        super.awakeFromNib()
    }
}
```

Code Sample 6.1

3. Add required UITableView methods

In the ConnectViewController.swift file we need to add the UITableViewDataSource and UITableViewDelegate protocols to process table data and confirm two required methods from UITableViewDataSource protocol (Code sample 6.2).

```
import UIKit

class ConnectViewController: UIViewController,
UITableViewDelegate, UITableViewDataSource {

    @IBOutlet weak var tableView: UITableView!

    override func viewDidLoad() {
        super.viewDidLoad()
        tableView.delegate = self
        tableView.dataSource = self
    }

    override func viewDidAppear(_ animated: Bool) {
        tableView.reloadData()
    }

    func tableView(_ tableView: UITableView, numberOfRowsInSection
section: Int) -> Int {
        return 0
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath:
IndexPath) -> UITableViewCell {
        if let cell = tableView.dequeueReusableCell(withIdentifier:
"deviceCell") as? DeviceCell {
```

```
            return cell
        }
        else {
            return DeviceCell()
        }
    }
}
```

Code sample 6.2

4. Preparing for Core Bluetooth

First, we need to add the CoreBluetooth framework with: import CoreBluetooth. Most of the work in the Core Bluetooth framework is done through delegate methods. The central is represented by CBCentralManager and its delegate is CBCentralManagerDelegate. CBPeripheral presents the peripheral device and its delegate is CBPeripheralDelegate.

To handle different states of the central device we need to add following extension to the ConnectViewController class (Code sample 6.3).

```
extension ConnectViewController: CBCentralManagerDelegate{
  func centralManagerDidUpdateState(_ central: CBCentralManager) {
    switch central.state {
    case .unknown:
      print("central.state is .unknown")
    case .resetting:
      print("central.state is .resetting")
    case .unsupported:
      print("central.state is .unsupported")
    case .unauthorized:
      print("central.state is .unauthorized")
    case .poweredOff:
      print("central.state is .poweredOff")
    case .poweredOn:
      print("central.state is .poweredOn")}}}
```

Code sample 6.3

Add to the ConnectViewController class the centralManager variable and make it initialization on viewDidLoad() method (Code sample 6.4). As the result the line "**central.state is .poweredOn**" will appear in console.

```
var centralManager: CBCentralManager!
centralManager = CBCentralManager(delegate: self, queue: nil)
```

Code sample 6.4

As the central device has entered the power on state (in case .poweredOn) it must scan for nearby peripherals with following code (Code sample 6.5).

```
        case .poweredOn:
    print("central.state is .poweredOn")
    centralManager.scanForPeripherals(withServices: nil)
```

Code sample 6.5

Now we need to discover the peripheral devices nearby implementing the following code (Code sample 6.6) in ConnectViewController extension. The result in console gives a list of devices that can be reached via Bluetooth, for instance: "<CBPeripheral: 0x2820a8000, identifier = F9D0E4DC-2FCE-372F-2358-C05E479DB9C8, name = Dmitriy's MacBook Pro, state = disconnected> <CBPeripheral: 0x2820a8000, identifier = DF68E247-B7C5-C285-6485-0D19ED04277A, name = iHealth Gluco, state = disconnected>"

```
func centralManager(_ central: CBCentralManager, didDiscover
peripheral: CBPeripheral, advertisementData: [String : Any], rssi RSSI:
NSNumber) {print(peripheral)}
```

Code sample 6.6

5. *Scanning for Peripherals with Specific Services*

We can scan for peripheral devices that provide only services that are

necessary for specific application, in this case – those devices which give information on glucose measurements. To do that, we need the UUID for the Glucose services (0x1808), which can be found on the Bluetooth services specification page https://www.bluetooth.com/specifications/gatt/services/ and note the UUID for it (Figure 6.4).

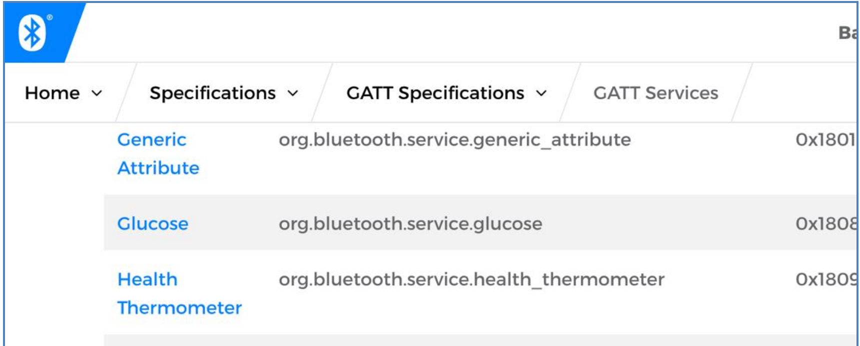


Figure 6.4

Now we need to create the CBUUID object and pass it to the scanForPeripherals(withServices:) that takes an array. The glucoseServiceCBUUID have to be placed under the import statements and referred from scanForPeripherals() method (Code sample 6.7).

```
let glucoseServiceCBUUID = CBUUID(string: "0x1808")

case .poweredOn:
print("central.state is .poweredOn")
        centralManager.scanForPeripherals(withServices:
    [glucoseServiceCBUUID])
```

Code sample 6.7

Next we need to store a reference to the glucose peripheral and then can stop scanning for further peripherals. To do that we need to create the glucosePeripheral variable and use the stopScan() method in centralManager(_:didDiscover: advertisementData:rssi:) (Code sample 6.8). After building and running an app we can find in console just one peripheral:

"<CBPeripheral: 0x2820a8000, identifier = DF68E247-B7C5-C285-6485-0D19ED04277A, name = iHealth Gluco, state = disconnected>".

```
var glucosePeripheral: CBPeripheral!

func centralManager(_ central: CBCentralManager, didDiscover
peripheral: CBPeripheral, advertisementData: [String : Any], rssi RSSI:
NSNumber) {
        print(peripheral)
        glucosePeripheral = peripheral
        centralManager.stopScan() }
```

Code sample 6.8

6. *Add Activity Indicator animation to the View Controller*

An Activity Indicator is a spinning wheel that indicates a task is being processed. If an action takes an unknown amount of time to process, we should display an activity indicator to let user know that app is not frozen. As the Activity Indicator starts working on Connect tab open and stops as all devices were found we need to implement the startSpinning() in viewDidLoad() and stopSpinning() after central.stopScan() method. The Code sample 6.9 presents @IBOutlet for Activity Indicator and startSpinning(), stopSpinnig() methods.

```
@IBOutlet weak var activityIndicator: UIActivityIndicatorView!

        func activityStart(){
                activityIndicator.startAnimating()}

           func activityStop(){
                activityIndicator.stopAnimating() }
```

Code sample 6.9

7. *Connecting to a peripheral*

To obtain data from a peripheral we need to connect it. Call the connect() method for centralManager after activityStop() and confirm the

connection by creating the centralManager(_:didConnect) delegate method (Code sample 6.10).

```
func centralManager(_ central: CBCentralManager, didDiscover
peripheral: CBPeripheral, advertisementData: [String : Any], rssi RSSI:
NSNumber) {
        print(peripheral)
        glucosePeripheral = peripheral
        centralManager.stopScan()
        self.activityStop()
        centralManager.connect(glucosePeripheral)
    }

    func centralManager(_ central: CBCentralManager, didConnect
peripheral: CBPeripheral) {
        print("Connected")
            }
```

Code sample 6.10

8. *Discovering a peripheral's services*

The next step after connection is to *discover* the *services* of the peripheral. Even after specifically requesting a peripheral with the glucose service we still need to *discover* the *service* to use it. After connecting, call the discoveryServices(nil) on the peripheral from centralManager(_:didConnect) delegate method. We can pass in UUID's for the services here, but for now we discover all available services to see what else the glucose device can do.

Next we need to implement the peripheral(_:didDiscoverServices:) delegate method. To do so we will create one more class extension to conform the CBPeripheralDelegate protocol (Code sample 6.11). The method peripheral(_:didDiscoverServices:) doesn't provide us a list of discoverable services but only that one or more services has been discovered by peripheral. This is because the peripheral object has a property that gives you a list of services**.**

```
extension ConnectViewController: CBPeripheralDelegate{
    func peripheral(_ peripheral: CBPeripheral, didDiscoverServices
```

```
error: Error?) {
      guard let services = peripheral.services else { return }

      for service in services{
         print(service)
      }
   }
}
```

Code sample 6.11.

Finally, point glucosePeripheral at its delegate with glucosePeripheral.delegate = **self** in centralManager(_:didDiscover: advertisementData:rssi:) and pass the glucoseServiceCBUUID to the glucosePeripheral.discoverServices() method. After building and running the application the following line will be printed to the console: <CBService: 0x1c046f280, isPrimary = YES, UUID = Glucose>**.**

9. Discovering a service's characteristics

The glucose measurement is a characteristic of a glucose service. To obtain the characteristics of a service we need to explicitly request the discovery of the service's characteristics. Add to the peripheral(_:didDiscoverService) the peripheral.discoverCharacteristics(nil, for:service).

After this implement peripheral(_:didDiscoveCharacteristicsFor:error:) after peripheral(_:didDiscoverDervices:). The CBPeripheralDelegate extension will look as in Code sample 6.12.

```
extension ConnectViewController: CBPeripheralDelegate{
   func peripheral(_ peripheral: CBPeripheral, didDiscoverServices
error: Error?) {
      guard let services = peripheral.services else { return }

      for service in services{
         print(service)
         peripheral.discoverCharacteristics(nil, for: service)
      }
   }
```

```
  func peripheral(_ peripheral: CBPeripheral,
didDiscoverCharacteristicsFor service: CBService,
                  error: Error?) {
      guard let characteristics = service.characteristics else { return }

      for characteristic in characteristics {
         print(characteristic)
      }
  }}
```

Code sample 6.12

Build and run the application. The console will show following information: <CBCharacteristic: 0x1c00b0920, UUID = 2A18, properties = 0x10, value = (null), notifying = NO> <CBCharacteristic: 0x1c00af300, UUID = 2A34, properties = 0x4, value = (null), notifying = NO>**.** On the Bluetooth specification page in the characteristics section we can see that 2A18 presents the glucose measurement and 2A34 shows the glucose measurement context (Figure 6.5). For these values we can add two constant values under the glucoseServiceCBUUID declaration (Code sample 6.13).

Home | Specifications | GATT Specifications | GATT Characteristics

Gender	org.bluetooth.characteristic.gender	0x2A8C	GSS
Glucose Feature	org.bluetooth.characteristic.glucose_feature	0x2A51	GSS
Glucose Measurement	org.bluetooth.characteristic.glucose_measurement	0x2A18	GSS
Glucose Measurement Context	org.bluetooth.characteristic.glucose_measurement_context	0x2A34	GSS

Figure 6.5

```
let glucoseMeasurementCharacteristicCBUUID = CBUUID(string:
"0x2A18")
let glucoseMeasurementContextCharacteristicCBUUID =
CBUUID(string: "0x2A34")
```

Code sample 6.13

10. Checking a characteristic's properties

Each characteristic has a property called properties of type CBCharacteristicsProperties and is an OptionSet. In this application we will focus on the .read only.

In peripheral(_:didDiscoverCharacteristicsFor:error:) method add code that helps to see the characteristics properties (Code sample 6.14). Build and run the application. We can see the result in console: 2A18: properties contain .read 2A34: properties contain .read. This means that both characteristics can let us read from them directly.

```
func peripheral(_ peripheral: CBPeripheral,
didDiscoverCharacteristicsFor service: CBService,
              error: Error?) {
      guard let characteristics = service.characteristics else { return }

      for characteristic in characteristics {
         print(characteristic)
         if characteristic.properties.contains(.read) {
            print("\(characteristic.uuid): properties contains .read")
         }
      }
        }
```

Code sample 6.14

11. Obtaining the Glucose Measurement data

The Core Bluetooth framework requires the implementation of peripheral(_:didUpdateValueFor:error:) method to read a characteristic's value. The read operation is asynchronous, which means that we request to read, and are then notified when the value has been read. Add peripheral(_:didUpdateValueFor:error:) to the CBPeripheralDelegate extension and peripheral.readValue(for:) in peripheral(_:didDiscoverCharacteristicsFor: error). The ConnectViewController extension is presented in Code sample 6.15.

```
extension ConnectViewController: CBPeripheralDelegate{
   func peripheral(_ peripheral: CBPeripheral, didDiscoverServices
error: Error?) {
      guard let services = peripheral.services else { return }
```

```
        for service in services{
            print(service)
            peripheral.discoverCharacteristics(nil, for: service)
        }
    }

    func peripheral(_ peripheral: CBPeripheral,
didDiscoverCharacteristicsFor service: CBService,
                    error: Error?) {
        guard let characteristics = service.characteristics else { return }

        for characteristic in characteristics {
            print(characteristic)
            if characteristic.properties.contains(.read) {
                print("\(characteristic.uuid): properties contains .read")
                peripheral.readValue(for: characteristic)
            }
        }
    }

    func peripheral(_ peripheral: CBPeripheral, didUpdateValueFor
characteristic: CBCharacteristic,
                    error: Error?) {
        switch characteristic.uuid {
        case glucoseMeasurementCharacteristicCBUUID:
            print(characteristic.value ?? "no value")
        case glucoseMeasurementContextCharacteristicCBUUID:
            print(characteristic.value ?? "no value")
        default:
            print("Unhandled Characteristic UUID: \(characteristic.uuid)")
        }
    } }
```

Code sample 6.15

1.6.4 Report requirements and tasks

Additional tasks:

1. Create the initial Connect page with empty table and UIButton that will segue to the ViewController described in section 1.6.3.

2. Perform all stages described in section 1.6.3 of searching, connecting and querying for data from connected peripheral device.

3. Display the names of found via Bluetooth appropriate devices to the table in Connect tab.

4. Call for default UIAlertController to ask user permission for connecting with peripheral devices.

5. Read information from selected peripheral and store it in the CoreData HistoryModel entity.

6. Store the name of connected device in Core Data and load all names to UITableView on initial Connect ViewController.

The report should contain following sections:

1. Introduction – background, theory and practical work purpose;

2. Development – the code with comments from ConnectViewController described in section 1.6.3 and InitialConnectViewController created while solving the task 1. Code and screenshots of all additional tasks solution with comments.

3. Summary – conclusion and result summary.

1.6.5 Test questions

1. What are the central and peripheral devices?

2. Describe two types of peripherals data?

3. How peripheral device can be found using CoreBluetooth framework?

4. How we can search for specific peripheral devices?

5. How we connect with peripheral devices, search for their services and characteristics?

1.6.6 Recommended literature

1. Core Bluetooth Framework Documentation. https://developer.apple.com/ documentation/ corebluetooth

2. Working With CoreBluetooth in iOS 11. Tutorial. https://www.appcoda.com/core-bluetooth/

3. Matt Neuburg. Programming iOS 12: Dive Deep Into Views, View Controllers and Frameworks/ o’Reilly Media, 2018 – 1176 p.

2. Developing IoT-based applications for Android

Practical work 2.1

GETTING STARTED WITH ANDROID STUDIO – INTRO TO THE DEVELOPMENT ENVIRONMENT

2.1.1 Synopsis

There aren’t any prerequisites for this practical work, other than a willing mind and a Mac or PC. You can develop for Android on both a Mac or a PC. The instructions mostly similar but slightly different between macOS, Windows and Linux.

You’ll learn how to set up all the tools needed to start you on your way to creating an Android application.

2.1.2 Brief theoretical information

Android Studio – is the official integrated development environment for Google's Android operating system, built on JetBrains' IntelliJ IDEA software and designed specifically for Android development. It is available for download on Windows, macOS and Linux based operating systems.

You can download Android studio on the following link: *https://developer.android.com/studio/index.html*.

2.1.3 Practical steps

1. Welcome screen and creating a project

You’ll start by creating a new Android app that you’ll use to explore Android Studio and to learn about its capabilities and interface. Fire up Android Studio and, in the Welcome to Android Studio window, select Start a new Android Studio project (Figure 1.1).

In the **Choose your project** window (Figure 1.2), there is bunch of possible options to choose from. We would be interested in the **Empty Activity** and **Bottom Navigation** Activity projects futher on. Also there are additional tabs on the top if you need to create an application for Wearables, Android TV, Android Auto or different other devices, which can be connected to Android through different channels.

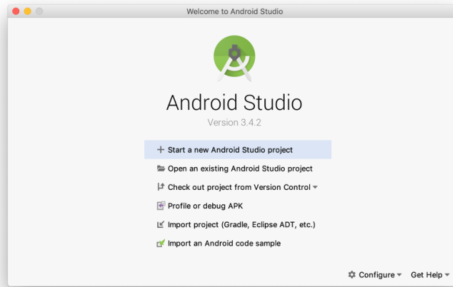


Figure 1.1. Android Studio – Welcome screen

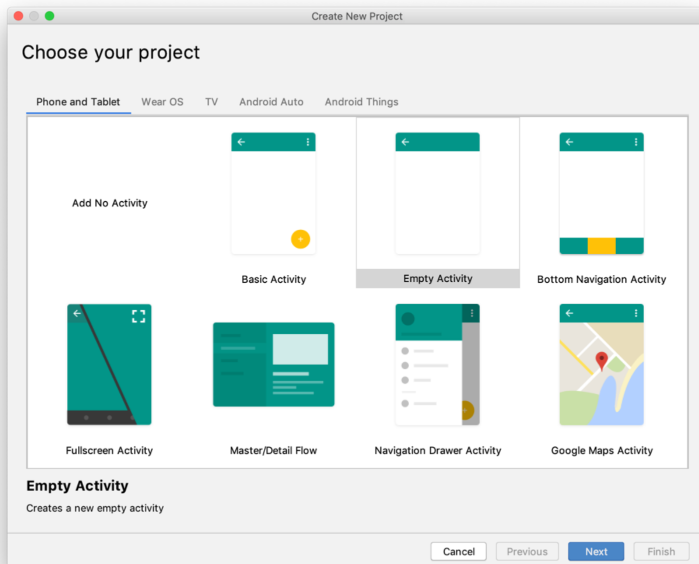


Figure 1.2. Android Studio – Welcome screen

After you choose a project type, you would need to fill the core fields for the application, which you can see on the Figure 1.3. Fields can be slightly different for different project types, but the main items are the same.

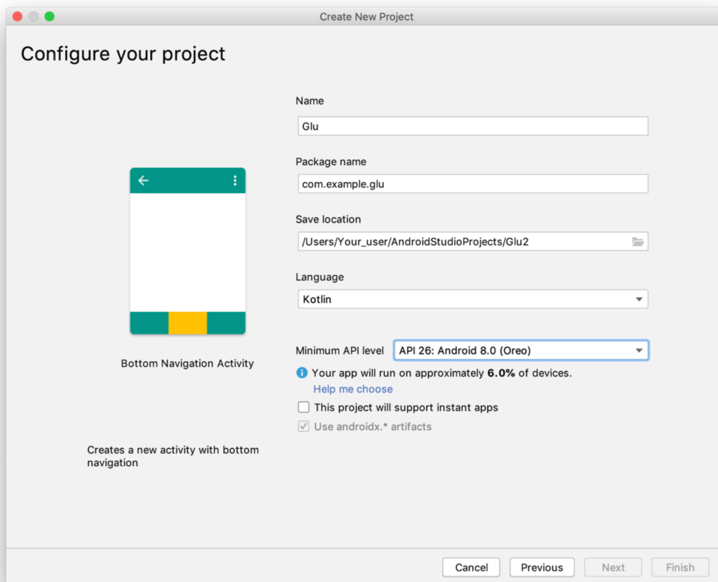


Figure 1.3. Android Studio – Configure screen

You are able to fulfill the following fields:

– **Name**: Your project actual name, you can pick Glu, or any other name

– **Package name**: Name of the package, Occasionally it's necessary to know the package name of an Android app. The package name is a unique name to identify a specific app. Generally, the package name of an app is in the format domain.company.application, but it's completely up to the app's developer to choose the name. The domain portion is the domain extension, like com or org, used by the developer of the app. The company portion is usually the name of the developer's company or product. The final application portion usually describes the app itself. This could be one word or multiple words separated by periods.

– **Save location:** Address of the folder location

– **Language:** You can choose the language to code, it would be Java or Kotlin in most of the cases. For our practical work we use Kotlin.

– **Minimum API Level:** Actual support of different android Versions, we would use the latest 28 version, but feel free to use anything after version 22 to have Kotlin support available.

After you press Finish and Within a short amount of time you’ll land on an application screen main UI, which would be your main screen for most of the time while working on any Android application.

2. Main window user interface

The Android Studio main window is made up of several logical areas identified in Figure 1.4.

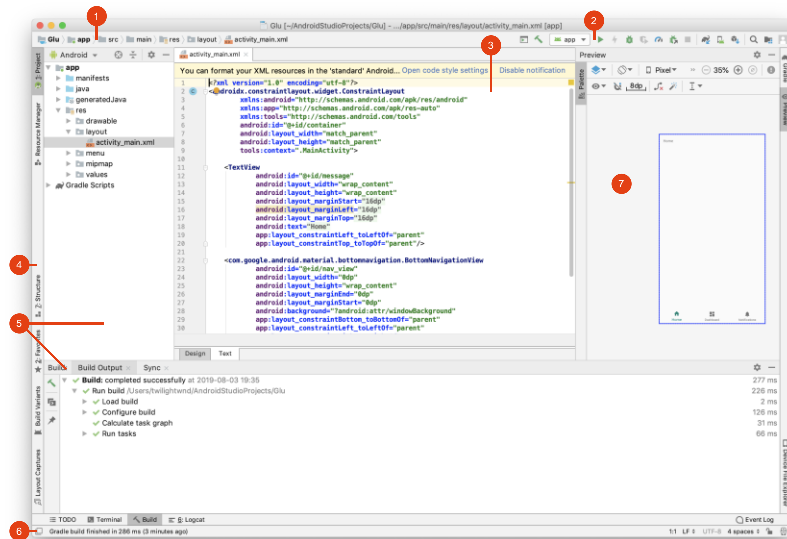


Figure 1.4. Android Studio – Main Window UI

– The **Navigation bar (1)** helps you navigate through your project and open files for editing. It provides a more compact view of the structure visible in the **Project** window.

– The **Toolbar (2)** lets you carry out a wide range of actions, including running your app and launching Android tools.

– The **Editor Window (3)** is where you create and modify code. Depending on the current file type, the editor can change. For

example, when viewing a layout file, the editor displays the Layout Editor.

– The **Tool Window Bar (4)** runs around the outside of the IDE window and contains the buttons that allow you to expand or collapse individual tool windows.

– The **Tool Windows (5)** give you access to specific tasks like project management, search, version control, and more. You can expand them and collapse them.

– The **Status Bar (6)** displays the status of your project and the IDE itself, as well as any warnings or messages.

– The **Preview window (7)** is one of the tool windows (5), but is particularly interesting for us, as it contains application UI. You can also switch between the code and preview windows in the development process.

You can organize the main window to give yourself more screen space by hiding or moving toolbars and tool windows. You can also use keyboard shortcuts to access most IDE features.

At any time, you can search across your source code, databases, actions, elements of the user interface, and so on, by double-pressing the Shift key, or clicking the magnifying glass in the upper right-hand corner of the Android Studio window. This can be very useful if, for example, you are trying to locate a particular IDE action that you have forgotten how to trigger.

3. Project Structure

Each project in Android Studio contains one or more modules with source code files and resource files.

Types of modules include:

– Android app modules

– Library modules

– Google App Engine modules

By default, Android Studio displays your project files in the Android project view, as shown in Figure 1. This view is organized by modules to provide quick access to your project's key source files.

All the build files are visible at the top level under Gradle Scripts and each app module contains the following folders:

– **manifests**: Contains the AndroidManifest.xml file.

– **java**: Contains the Java source code files, including JUnit test code.

– **res**: Contains all non-code resources, such as XML layouts, UI strings, and bitmap images

The Android project structure on disk differs from this flattened representation. To see the actual file structure of the project, select Project from the Project dropdown (in Figure 1.5, it's showing as Android).

You can also customize the view of the project files to focus on specific aspects of your app development. For example, selecting the Problems view of your project displays links to the source files containing any recognized coding and syntax errors, such as a missing XML element closing tag in a layout file.

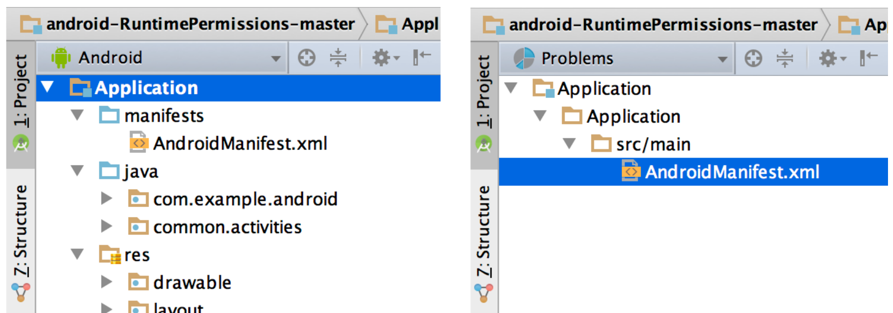


Figure 1.5 Android product structure

4. Version control basics

Android Studio supports a variety of version control systems (VCS’s), including Git, GitHub, CVS, Mercurial, Subversion, and Google Cloud Source Repositories.

After importing your app into Android Studio, use the Android Studio VCS menu options to enable VCS support for the desired version control system, create a repository, import the new files into version control, and perform other version control operations:

From the Android Studio VCS menu, click Enable Version Control Integration.

From the drop-down menu, select a version control system to associate with the project root, and then click OK.

The VCS menu now displays a number of version control options based on the system you selected.

You can read more about **git** as an example in iOS part of the practical work.

2.1.4 Report requirements and tasks.

There aren’t any prerequisites for this practical work, other than a willing mind and a Mac or PC.

1. Using the links in the practical work successfully install the Android studio.
2. Get an overview of the tool, using the steps of the practical work.
3. Write down questions, if there are any left.

2.1.5 Test questions.

1. What types of template projects does Android studio provide?
2. Describe the core parts of the project structure.
3. What is VCS? Name at least 3 of the most popular ones.
4. Which tool window can you use for application UI?

Practical work 2.2

DESIGN AND BASIC LAYOUTS OF THE ANDROID DIABETIC TRACKER APPLICATION “GLUCOSE”

2.2.1 Synopsis

This practical work presents a pointed analysis of how the Material Design patterns and guidelines can be applied to design a health-related application

2.2.2 Brief theoretical information

The Google Play Market gives a list of recommendations about content policy that should be fulfilled if application is going to be published in Android applications market. It should be noted that recommendations are less strict than presented by App Store Review team, but still highlight the most important issues such as violent content restrictions, handling user data, monetization plans, advertisement etc. While there are no specific requirements for working with user health data we can steel use information on how app must handle sensitive user data [1]:

- limit collections and use this data to purpose directly to providing and improving the feature of the app;
- post a privacy policy that comprehensively disclose how app collects, uses and shares user data;
- handle all personal data securely, including transmitting it using modern cryptography.

Additionally, Google Play Market set no specific recommendation on application design that is planned to be submitted to the store. Still, starting from 2014 Google develop a specific design language, known as Material Design. Material is an adaptable system of guidelines, components and tools that support the best practices of user interface design. Material Design can be used in all supported versions of Android and Google has also released APIs for third-party developers to incorporate the design language into their applications. Thus, the Android diabetic tracker application “Glucose” design will be built upon Material design approach.

Now, let’s list the basic functions that glucose tracker application will provide:

- manually add new glucose measurement: set data in ml/dg, dependence on meal, date and time;
- synchronize and app with third-party glucometers to upload the recent data;
- present a glucose measurement history to the user;
- edit the glucose measurement history data;
- send reminders for the next measurement time.

2.2.3 Practical steps

1. Measure page design

First, let's discuss how basic application pages can be organized. There are four logical groups can be derived from apps functions listed above – management of the history data (History); creation of a new measurement (Measure); connection to the third-party devices (Connect) and user notifications (Settings). We can present this data with lateral navigation that refers to movement between screens at the same hierarchy level.

According to Material Design essentials this type of navigation can be created with bottom navigation bar in case if there are 2 – 5 top-level destinations and application is developed for mobile device. The bottom navigation must be ergonomic, consistent and present only equally important items. In case of four destinations both active and inactive items should be presented with icons and titles and have sufficient contrast with the container [2].

The Figure 2.1 presents Measure tab that let user add following information to an app: ongoing glucose measure, dependence of this data on meal and date with time. There first two inputs are required but the last one can be optional. As the new data record is created the system will simply use the present time.

There are three options for meal selection: Before, After and Bedtime which can be accessed through filled exposed dropdown menu (Figure 2.2).

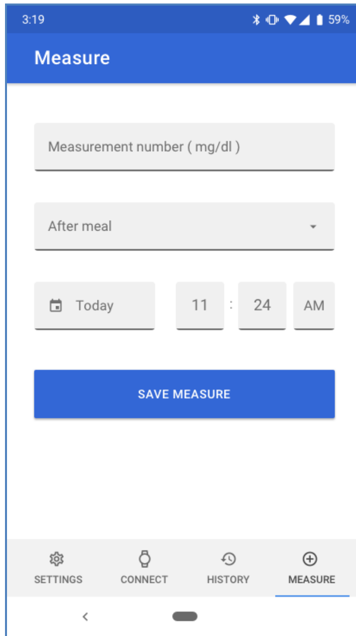


Figure 2.1

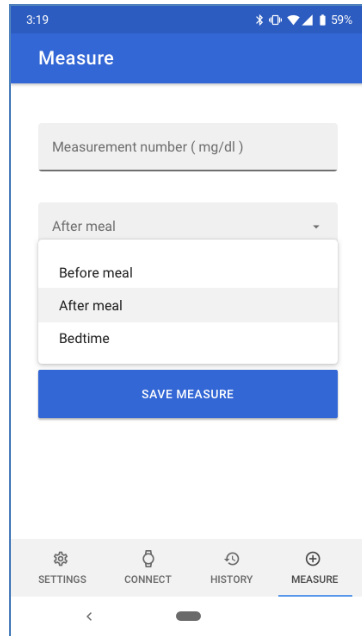


Figure 2.2

Menu items should be easy to open, scan, close and interact with. Menu height should be at least one row less than the height of app's UI. This item typically appears next to the element that generates them. The filled exposed dropdown menu displays the currently selected menu item above the menu and applied only when a single variant can be chosen at a time.

Date selection can be organized with date picker element that is activated with date picker field. This control can display past, present or future dates based on task relevance, clearly indicate important dates and ensure picking a day or time is intuitive. For our purpose the classic Material Design mobile calendar date picker is the most suitable one (Figure 2.3). The time picker can be organized using the filled text fields. There are three main principles for text fields design: easy to discover, clearly differentiate from one another and efficient.

2. History page design

The previously glucose measurement history can be organized in

a table form (Figure 2.4). The grid-like format or rows and columns is one of the most essential for presenting such data sets. Material Design guidelines draw three main principles for data tables design: organize internal content in meaningful way (hierarchy or alphabetization); allow user interactions for additional user customization; easy to use with clear logical structure [3].

The History tab presents a simple table that contains glucose measure information in clear and readable way. Each row presents a previously made glucose record with a dropdown menu with “Delete” option.

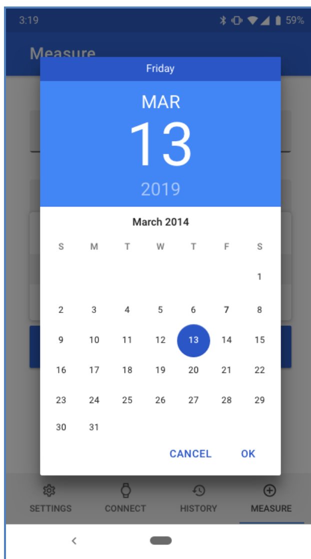


Figure 2.3

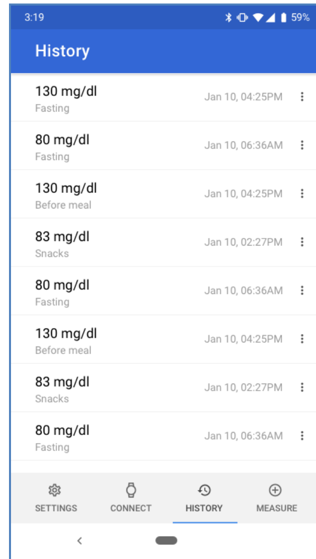


Figure 2.4

3. Connect page design

According to the apps functions list, the user has an option to connect through Bluetooth some third-party devices to read the most recent data. Figure 2.5 presents an initial page in Connect tab that provide an editable list of recently connected devices. New connection can be established with “Add device” button located in page bottom section. This button will take user to the next screen where search of new device is performed (Figure 2.6). All buttons under Material Design

guidelines should be highly identifiable, easy to find and present clear actions. In this app we use the contained button type as it has more emphasis while using the color and shadow.

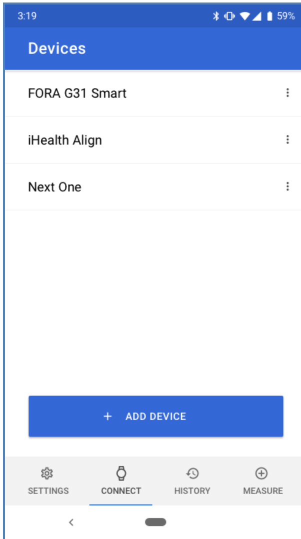


Figure 2.5

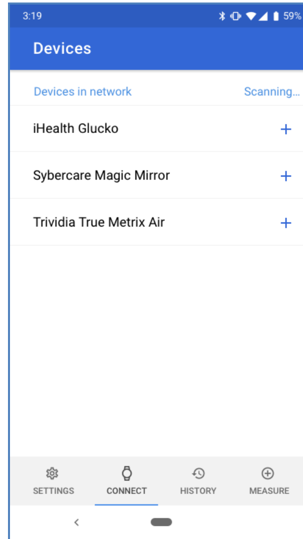


Figure 2.6

We can show a search feedback to user with updated list of nearby devices that have turned on Bluetooth module and loader "Scanning…" that will animate during search. User can connect to the needed device by simply clicking on "+" in the row near its name. As the result the dialog alert window will appear that asks to confirm paring to device (Figure 2.7). Dialog components are of high-priority components, which mean that it will block app usage until the user takes a dialog action or exits the dialog. Based on this the dialogs should be used carefully and applied for handling the critical information that requires a specific user tasks, decisions or acknowledgement.

4. Settings page design

The developed application presents only one additional setup that can be made by user – set reminders on the next glucose measurement (Figure 2.8). The notifications may be noticed by user by

showing a status bar icon, appearing on the lock screen, playing a sound or vibrating, peeking onto the current screen or blinking the device's LED. Android platform guidance set a list of information when notifications should not be used and when they should [4] .

From the Settings page notifications can be enabled with switch control. When user toggles a switch, its corresponding action takes effect immediately. If a switch cannot be turned on, the switch should automatically turn back off letting the user know that it is unavailable.

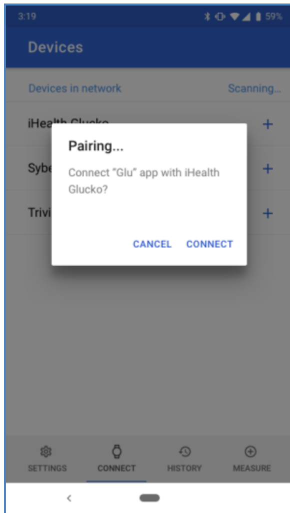


Figure 2.7

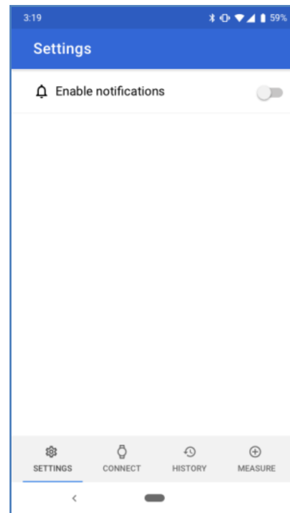


Figure 2.8

2.2.4 Report requirements and tasks

Practical work tasks:

4. Download Sketch or Figma, install the software and design the basic «Glucose» application screens. You can use the partial or full design and data organization of «Glucose» application as it was presented in 2.2.3 Practical steps. Use thenounproject.com and material.io to find icons for buttons and other control elements.

5. Read the official Material Design guidelines following elements: buttons, labels, date pickers, switchers, text fields, dialogs and tables.

6. Add into the Settings tab following additional setup functions: select the glucose units from mg/dL to mmol/L; clear measurements history; delete measurement history for data later than month ago; setup reminder with custom settings inside an application.

The report should contain following sections:

10. Introduction – background, theory and practical work purpose;

11. Development – screenshots with explanation of each practical work task completion.

12. Summary – conclusions and result summary.

1.2.5 Test questions

1. What is a Material Design?
2. What are the basic Material Design requirements for layout organization?
3. What type of information is presented in Measure screen? Why did you used such controls to get user data?
4. What type of information is presented in History screen? How we can alternate the data presentation in this screen?
5. Name the representation stages of searching and connecting to the peripheral device in Connect page.

1.2.6 Recommended literature and resources

1. Developer Content Policy. https://play.google.com/intl/en_us/about/developer-content-policy/
2. Bottom Navigation. https://material.io/design/components/bottom-navigation.html
3. Data Tables. https://material.io/design/components/data-tables.html#
4. Android Notifications. https://material.io/design/platform-guidance/android-notifications.html#

Practical work 2.3

TRANSLATING DESIGN INTO CODE - ADD AND SETUP BASIC “GLUCOSE” FRAGMENTS

2.3.1 Synopsis

In this practical work, we’ll learn how to use the basic components of Android application, such as lists, bottom navigation, different input fields and datepicker. We would use the material component library to make it easier.

i. Brief theoretical information

Material Components for Android (MDC Android) unites design and engineering with a library of components for creating consistency across your app. As the Material Design system evolves, these components are updated to ensure consistent pixel-perfect implementation and adherence to Google's front-end development standards.

You can check any additional information about the material components and guidelines, native android components and on [1]. Additional guides for different components are presented at [2].

There are libraries, which allows to simplify design to development process by replacing XML files, one of those is Anko. Anko is a Kotlin library [3], which makes Android application development faster and easier. It makes your code clean and easy to read.

ii. Practical steps

1. Create a project

Create a project (Figure 3.1), similar to what you did in the first practical work. Use the template for bottom navigation.

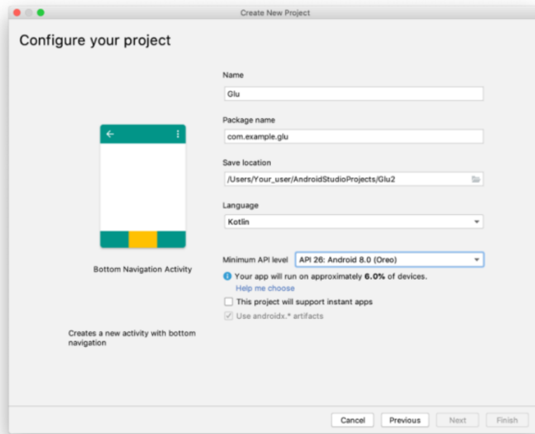


Figure 3.1. Android Studio – Configure your project

2. Setting up a bottom navigation bar

To use Anko and Navigation Architecture Component on module level we need to implement dependencies in build.gradle.

```
implementation
"org.jetbrains.anko:anko:$anko_version"
implementation "org.jetbrains.anko:anko-
constraint-layout:$anko_version" implementation
"com.android.support.constraint:constraint-
layout:2.0.0-alpha3"
implementation
'android.arch.navigation:navigation-
fragment:1.0.0-beta02'
implementation
'android.arch.navigation:navigation-fragment-
ktx:1.0.0-beta02'
implementation
'android.arch.navigation:navigation-ui-
ktx:1.0.0-beta02'
implementation
'com.google.android.material:material:1.0.0'
```

Next step is to create the structure of our application. To draw the main activity instead of xml-file we can create MainActivityUI, based on the AnkoComponent class. Therefore, it would be an override of the AnkoComponent class.

```
class MainActivityUI:
AnkoComponent<MainActivity> {
    override fun createView(ui:
AnkoContext<MainActivity>): View = with(ui)
{
        constraintLayout {  }
    }
}
```

Replace MainActivity setContentView(R.layout.activity_main) with MainActivityUI().setContentView(this) in the MainActivity class.

Then we create package fragments which would contain our fragments and package ui for the classes, which are used for drawing the screens of the corresponding fragments. Here is how the structure would look like, according to the design:

```
fragments
        ui
            SettingsUI
            ConnectUI
            HistoryUI
            MeasureUI
        SettingsFragment
        ConnectFragment
        HistoryFragment
        MeasureFragment
```

Now we would start actually working on Navigation and creation of Bottom Navigation Bar. Detailed guide on how to add new navigation components and Navigation Editor work guide you can find at [4].

To create a graph navigation file between the application screens, we would need to add an additional folder, named **navigation** into the **res** folder and create the **navigation_graph.xml** inside.

```
<navigation
xmlns:android="http://schemas.android.com/apk/re
s/android"
xmlns:app="http://schemas.android.com/apk/res-
auto"
    android:id="@+id/navigation_graph"
app:startDestination="@id/settingsFragment">

    <fragment
        android:id="@+id/settingsFragment"
android:name="com.arsinde.ankobottomnavbar.fragm
ents.SettingsFragment"
        android:label="SettingsFragment">
        <action
android:id="@+id/action_settingsFragment_to_meas
ureFragment"

app:destination="@id/measureFragment"/>
    </fragment>

    <fragment
        android:id="@+id/historyFragment"
android:name="com.arsinde.ankobottomnavbar.fragm
ents.HistoryFragment"
        android:label="HistoryFragment">
        <action
android:id="@+id/action_historyFragment_to_conne
ctFragment"

app:destination="@id/connectFragment"/>
    </fragment>

    <fragment
        android:id="@+id/connectFragment"

android:name="com.arsinde.ankobottomnavbar.fragm
ents.ConnectFragment"
        android:label="ConnectFragment">
```

```
        <action
android:id="@+id/action_connectFragment_to_measu
reFragment"

app:destination="@id/measureFragment"/>
    </fragment>

    <fragment
        android:id="@+id/measureFragment"
android:name="com.arsinde.ankobottomnavbar.fragm
ents.MeasureFragment"
        android:label="MeasureFragment"/>
</navigation>
```

For the current project we would need four items, based on design. Also we are using the default material icons which were taken from [5].

We would need to create another resource folder to show the Bar itself, which would be responsible for visuals view of the bar. Here is how it would look like:

```
<menu
xmlns:android="http://schemas.android.com/apk/re
s/android">
    <item
           android:id="@id/settingFragment"
           android:icon="@drawable/ic_settings"
android:title="@string/menu_title_settings"
/>
    <item
android:id="@id/connectFragment"
android:icon="@drawable/ic_watch"
android:title="@string/menu_title_history"
/>
    <item
android:id="@id/historyFragment"
android:icon="@drawable/ic_history"
android:title="@string/menu_title_measure"
/>
```

```
    <item
android:id="@id/measureFragment"
android:icon="@drawable/ic_add"
android:title="@string/menu_title_more"
/>
</menu>
```

Now we can connect everything together and check how it works. We would add the container for fragments into the MainActivityUI, and define the container for navigation bar

```
</menu> constraintLayout {
        val fragmentContainer = frameLayout {
            id = R.id.fragment_container
        }.lparams {
            width = matchParent
            height = matchConstraint
        }
        val bottomNavigation = bottomNavigation
    {
            id = R.id.bottom_nav_view

    inflateMenu(R.menu.bottom_navigation_menu)
        }
        applyConstraintSet {
            fragmentContainer {
                connect(
                START to START of PARENT_ID,
                END to END of PARENT_ID,
                TOP to TOP of PARENT_ID,
                BOTTOM to TOP of
    R.id.bottom_nav_view
                )}
            bottomNavigation {
                connect(
                    START to START of PARENT_ID,
                    END to END of PARENT_ID,
                    TOP to BOTTOM of
    R.id.fragment_container,
```

```
                BOTTOM to BOTTOM of
PARENT_ID
            )}}}
```

Also keep in mind, that bottomNavigation in this practical work is an extension function:

```
inline fun ViewManager.bottomNavigation(init: BottomNavigationView.() -> Unit = {}) =
        ankoView({ BottomNavigationView(it) }, theme = 0, init = init)
```

Now we need to define NavHostFragment in MainActivity:

```
private val host by lazy { NavHostFragment.create(R.navigation.navigation_graph) }
```

And define it in onCreate():

```
supportFragmentManager.beginTransaction()
    .replace(R.id.fragment_container, host)
    .setPrimaryNavigationFragment(host)
    .commit()
```

The last step in creating navigation is adding the NavControllel class object into onStart() MainActivity, which would make the switch between the fragments, choosing the corresponding object in navigation bar

```
override fun onStart() {
    super.onStart()
    val navController = host.findNavController()

findViewById<BottomNavigationView>(R.id.bottom_nav_view)?.setupWithNavController(navController)

navController.addOnDestinationChangedListener{_, destination, _ ->
        val dest: String = try {

resources.getResourceName(destination.id)
```

```
        } catch (e: Resources.NotFoundException)
{
            Integer.toString(destination.id)
        }
        Log.d("NavigationActivity",    "Navigated
to $dest")
    }}
```

Here’s how the basic version would look as presented on Figuew 3.2.

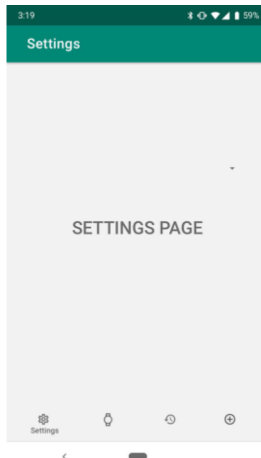


Figure 3.2. Bottom navigation

3. Datepicker setup

We would leave the easier parts for the tasks in the end, and would check the most complex component, among those we have in our app, the datepicker:

Create a fresh project. Add the following code into the **activity_main.xml** layout file, where button have the method to perform onClick action

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/re
s/android"

xmlns:app="http://schemas.android.com/apk/res-
auto"

xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:text="Hello World!"

app:layout_constraintLeft_toLeftOf="parent"

app:layout_constraintRight_toRightOf="parent"

app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
        android:text="Open Date Picker"
        android:onClick="clickDataPicker"

app:layout_constraintEnd_toEndOf="parent"

app:layout_constraintStart_toStartOf="parent"
app:layout_constraintTop_toBottomOf="@+id/textVi
ew" />
</android.support.constraint.ConstraintLayout>
```

Setup the MainActivity.kt. On clicking button – Creates a new date picker dialog for the current date using the parent context’s default date picker dialog theme (Figure 3.3). Context is requires the application context.

var year: It shows the the current year that’s visible when the dialog pops up

var month: It shows the the current month that’s visible when the dialog pops up

var dat: It shows the the current day that’s visible when the dialog pops up

```
package `in`.eyehunt.androiddatepickerdialog

import android.app.DatePickerDialog
import android.icu.util.Calendar
import android.os.Build
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import android.support.annotation.RequiresApi
import android.view.View
import android.widget.Toast

class MainActivity : AppCompatActivity() {

    override   fun   onCreate(savedInstanceState: 
Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }

    @RequiresApi(Build.VERSION_CODES.N)
    fun clickDataPicker(view: View) {
        val c = Calendar.getInstance()
        val year = c.get(Calendar.YEAR)
        val month = c.get(Calendar.MONTH)
        val day = c.get(Calendar.DAY_OF_MONTH)

```

```
        val    dpd    =    DatePickerDialog(this,
DatePickerDialog.OnDateSetListener { view, year,
monthOfYear, dayOfMonth ->
            // Display Selected date in Toast
            Toast.makeText(this,   """$dayOfMonth
-    ${monthOfYear    +    1}    -    $year""",
Toast.LENGTH_LONG).show()

        }, year, month, day)
        dpd.show()
    }
}
```

Figure 3.3. Date picker after implementing to the system

iii. Report requirements and tasks

1. Using the links and code in the implement the navigation menu.
2. Using the links and code implement the datepicker.

3. Check material library and android guides at https://developer.android.com/docs and https://material.io/develop/
4. Adapt the design colors using XML or Anko
5. Implement any 2 other features, which were done in design (inputs and settings for example)

iv. Test questions.

1. How do you create the package fragments?
2. What are the core variables for datepicker?
3. What is the advantage of using Anko over XML?

v. Literature

1. Material Design. Develop. https://material.io/develop/
2. Material Design. Developer Tutorials. https://material.io/collections/developer-tutorials/#android-kotlin
3. Anko. https://github.com/Kotlin/anko
4. Developers. BottomNavigationView. https://developer.android.com/reference/android/support/design/widget/BottomNavigationView
5. Material Design. Icons. https://material.io/resources/icons/

Practical work 2.4

GETTING STARTED WITH DATABASES ON ANDROID

2.4.1 Synopsis

In this practical work we will discuss how user data can be stored on Android device on the example of SQLite database. Starting with creation of a new database, insert of a new record into it, updating information, editing existed data and deleting its content we will learn the basic operations with SQLite database.

2.4.2 Brief theoretical information

There are several options that can be applied to store user data:

- Internal file storage – store application files on the device;
- External file storage – store files on the shared external file system;
- Shared preferences – store private primitive data in key-value pairs;
- Databases – store structured data in a private database.

As it was stated before, the application stores glucose measurements along with meal dependence data and date with time as it was made, thus the databases will be a good place where to store this information.

Android provides full support of SQLite databases that will be only accessible from app that have created it. However, the official documentation suggests to communicate with database with Room persistence library instead simple SQLite APIs. The Room library provides an object-mapping abstraction layer that allows fluent databases access and thus takes care of many concerns. The Android SDK includes a sqlite3 database tool that allows browsing table contents, run SQL commands etc [1].

2.4.3 Practical steps

1. Creating a Measure.java class

Create several packages, namely database and database/model. Inside of database/model packages create a Measure.java (Code sample

4.1) that will define the SQLite table and column names and create table SQL query along with get and set methods.

The **“measure”** table have five columns:

- **“id”** column is defined as Primary Key and Auto Increment which means that every measurement record will be uniquely identified by its id;

- **“glucose”** stores a string with glucose measurement data from the text field;

- **“meal”** stores a string with definition of meal dependence selected from the dropdown menu;

- **“date”** stores string created from Date Picker and time entered from text fields. This data is optional, thus if user did not set this information here will be stored data and time of when the new record was created;

- **“timestamp”** stored the data and time of the record that is created. This data can help in refreshing table if the user has edited its data.

```
public class Measure {
    public static final String TABLE_NAME =
"measure";
    public static final String COLUMN_ID = "id";
    public static final String COLUMN_GLUCOSE =
"glucose";
    public static final String COLUMN_MEAL =
“meal”;
    public static final String COLUMN_DATE =
“date”;
    public static final String COLUMN_TIMESTAMP
= "timestamp";

    private int id;
    private String glucose;
    private String meal;
    private String date;
    private String timestamp;

    public static final String CREATE_TABLE =
```

```
    "CREATE TABLE " + TABLE_NAME + "("
    + COLUMN_ID + " INTEGER PRIMARY KEY
AUTOINCREMENT,"+ COLUMN_GLUCOSE + " TEXT,"
    + COLUMN_MEAL + " TEXT,"+ COLUMN_DATE + "
TEXT,"+ COLUMN_TIMESTAMP + " DATETIME DEFAULT
CURRENT_TIMESTAMP"+ ")";

    public Measure() {}
    public Measure(int id, String glucose,
String meal, String date, String timestamp) {
        this.id = id;
        this.glucose = glucose;
        this.meal = meal;
        this.date = date;
        this.timestamp = timestamp;
    }
    public int getId() { return id; }
   public void setId(int id) { this.id = id; }
   public String getGlucose() { return glucose;}
   public void setGlucose(String glucose) {
        this.glucose = glucose; }
    public String getMeal() { return meal; }
    public void setMeal(String meal) { this.meal
= meal; }
    public String getDate() { return date; }
    public void setDate(String date) {
        this.date = date; }
    public String getTimestamp() {
        return timestamp;}
    public void setTimestamp(String timestamp) {
        this.timestamp = timestamp;}
```

Code sample 4.1

2. Creating SQLite Helper class

First we need to create the SQLite helper class (DatabaseHelper.java) in database package, that will extend from SQLiteOpenHelper (Code sample 4.2) [2]. The SQLiteOpenHelper is a helper class for managing database creation and version management.

The DatabaseHelper.java will implement onCreate(SQLiteDatabase), ouUpgrade(SQLiteDatabase, int, int) methods.

The onCreate() is called only once when the app is installed as this method executes and the sql statement which creates a table.

The onUpdate() will be called when an update is released.

```
public class DatabaseHelper extends
SQLiteOpenHelper {
     private static final int DATABASE_VERSION =
1;
     private static final String DATABASE_NAME =
"measure_db";

    public DatabaseHelper(Context context) {
        super(context, DATABASE_NAME, null,
DATABASE_VERSION);
    }
           // Create Tables
    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(Measure.CREATE_TABLE);
    }
          // Upgrade database
    @Override
    public void onUpgrade(SQLiteDatabase db, int
oldVersion, int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS " +
Measure.TABLE_NAME);
        onCreate(db);
    } }
```

Code sample 4.2

3. Insert data to Measure database table

New data insertion requires getting the writable instance with getReadableDatabase() on database. The ContentValue() is applied to define column name and data that it stores. The “id” and “timestamp” columns do not require setting up as these two will be inserted

automatically. It should be noted that database connection have to be closed with db.close() as soon the following work does not require the database use. As soon as glucose, meal and data values are inserted, the "id" of new insertion will be returned. The code sample 4.3 presents method for new data insertion to the database.

```
  public long insertMeasure(String measure) {

      SQLiteDatabase db =
this.getWritableDatabase();
      ContentValues values = new ContentValues();
      values.put(Measure.COLUMN_GLUCOSE,
glucose);
      values.put(Measure.COLUMN_MEAL, meal);
      values.put(Measure.COLUMN_DATA, data);

    long id = db.insert(Measure.TABLE_NAME,
null, values);
    db.close();
    return id;
}
```

Code sample 4.3

4. Reading data from Measure database table

With the same public method of SQLiteOpenHelper class getReadableDatabase() we can open a database for reading it data. The code sample 4.4 presents an application of getAllMeasures() method that fetches all measures in descending order by timestamp. It returns an ArrayList that can be further used for presenting data in History tab table.

```
    public List<Measure> getAllMeasures() {
     List<Measure> measure = new ArrayList<>();

    // Select All Query
    String selectQuery = "SELECT  * FROM " +
Measure.TABLE_NAME + " ORDER BY " +
Measure.COLUMN_TIMESTAMP + " DESC";
```

```
    SQLiteDatabase db =
this.getWritableDatabase();
    Cursor cursor = db.rawQuery(selectQuery,
null);
    if (cursor.moveToFirst()) {
        do {
    Measure measure = new Measure();
    measure.setId(cursor.getInt(cursor.getColumn
Index(Measure.COLUMN_ID)));

measure.setGlucose(cursor.getString(cursor.getCo
lumnIndex(Measure.COLUMN_GLUCOSE)));
     measure.setMeal(cursor.getString(cursor.get
ColumnIndex(Measure.COLUMN_MEAL)));
     measure.setData(cursor.getString(cursor.get
ColumnIndex(Measure.COLUMN_DATA)));
      measure.setTimestamp(cursor.getString(curs
or.getColumnIndex(Measure.COLUMN_TIMESTAMP)));
            measure.add(measure);
        } while (cursor.moveToNext());
    }
    db.close();
    return measure;
}
```

Code sample 4.3

5. Updating data from Measure table database

The data update requires the writable access provided with getWritableDatabase(). In code sample 4.4 the data is updated using its “id”.

```
public int updateMeasure(Measure measure) {
    SQLiteDatabase              db              =
this.getWritableDatabase();

    ContentValues values = new ContentValues();
    values.put(Measure.COLUMN_GLUCOSE,
```

```
measure.getGlucose());
    values.put(Measure.COLUMN_MEAL,
measure.getMeal());
    values.put(Measure.COLUMN_DATA,
measure.getData());

    return  db.update(Measure.TABLE_NAME,  values,
Measure.COLUMN_ID + " = ?",
        new
String[]{String.valueOf(measure.getId())});}
```

Code sample 4.4

6. Deleting data from Measure table database

The same as above, if we need to delete data from databases there should be writable access to it. Method presented in Code sample 4.5 deletes a measure by finding its “id”.

```
public void deleteMeasure(Measure measure) {
    SQLiteDatabase db =
this.getWritableDatabase();
    db.delete(Measure.TABLE_NAME,
Measure.COLUMN_ID + " = ?",
   new
String[]{String.valueOf(Measure.getId())});
    db.close();
}
```

Code sample 4.5

2.4.4 Report requirements and tasks

Practical work tasks:

6. Create a new SQLite database in existing glucose measurement application project.
7. Apply the insertMeasure(string measure) method on “Add measure” button pressed event.
8. Present the ArrayList returned with getAllMeasures() method inside of table in History tab.

9. Using updateMeasure() and deleteMeasure() methods organize the according editing options for table in History tab.

The report should contain following sections:

13. Introduction – background, theory and practical work purpose;
14. Development – screenshots with explanation of each practical work task completion; DatabaseHelper.java class code with comments; History tab classes code with comments.
15. Summary – conclusions and result summary.

2.4.5 Test questions

1. How user data can be stored on Android devices?
2. Explain the difference between different Android storages?
3. What methods are applied to make a new data record to the SQLite?
4. How we can read, update, delete and edit data from SQLite?

2.4.6 Recommended literature and resources

1. Data and File Storage Overview. https://developer.android.com/guide/topics/data/data-storage#db
2. android.database.sqlite Documentation. https://developer.android.com/reference/android/database/sqlite/package-summary

Practical work 2.5

INTEGRATING THIRD - PARTY TRACKERS AND GLUCOMETERS USING API.

2.5.1 Synopsis

In this practical work we'll learn how to use the third party trackers and glucometers using API based on the example of iHealth API.

2.5.2 Brief theoretical information

iHealth is a healthcare management company striving to revitalize old healthcare devices with modern technology everyone is familiar with. The MyVitals and Gluco-Smart Mobile App can synchronize with all iHealth products and allow you to view every result in one app. It also have simple custom API for the third-party applications, so everyone can track the progress as a developer, using iHealth devices.

There are also multiple other companies, which devices can be use with the same idea in mind: GlucoWise, DarioHealth, Abbott Diabetes Care, Integrity Applications, Senseonics etc.

2.5.3 Practical steps

1. Direct methods

iHealth Wireless Smart Gluco-Monitoring System have the basic API Calls, which are pretty simple, they are:

- **BG_GET** for getting the data, using the following query:

```
sc
sv
client_id
client_secret
```

- **BGALL_GET** for getting the full set of data, using the following query:

```
sc
sv
client_id
client_secret
```

- **BG_POST** for sending the data to device, using the following body items:

```
MDATE
TIMEZONE
BG
DINNERSITUATION
DRUGSITUATION
BGUNIT
```

- **BG_PUT** for putting the data, using the following body items:

```
MDATE
TIMEZONE
BG
DINNERSITUATION
DRUGSITUATION
BGUNIT
```

- query:

```
sc
sv
```

We can use this information to work with basic API.

2. *Complex integrations using SDK*

For more complex integrations or seamless work with different devices it is more reliable to use an SDK. There are some examples of code for iOS on official iHealth github, which you can check and adapt to Android, as iHealth does not have official guides for Android yet [1, 2].

We can use an Android SDK from [3] to work further.

iHealth Device SDK can accomplish the major operations such as: Connection Device, Online Measurement, Offline Measurement and iHealth Device Management. To start using SDK you would need to initialize it first, you can do it with the following code.

```
iHealthDevicesManager.getInstance().init(MainActivi
ty.this);
```

To register a callback and get a callback ID use:

```
int callbackId =
iHealthDevicesManager.getInstance().registerClientC
allback(iHealthDevicesCallback);
```

We can also use a callback filter in the similar way:

```
iHealthDevicesManager.getInstance().addCallbackFilt
erForAddress(clientCallbackId, ...);
iHealthDevicesManager.getInstance().addCallbackFilt
erForDeviceType(clientCallbackId, ...);
```

Now let’s get to devices itself: To verify the iHealth device user permission you would need to use the following code:

```
iHealthDevicesManager.getInstance().sdkUserInAuthor
(MainActivity.this, userName, clientId,
clientSecret, callbackId);
```

If verify success, all the api avaliable, else you will get 10 trial day, and would need to contact the iHealth to get the developer license for it. But we can use the trial for the learning purposes.

Now we would need to discover the device:

```
int type = iHealthDevicesManager.DISCOVERY_BP5
    iHealthDevicesManager.getInstance().startDiscov
ery(type);

    private iHealthDevicesCallback
iHealthDevicesCallback = new
iHealthDevicesCallback() {
     @Override
        public void onScanDevice(String mac, String
deviceType) {
        }
    };
```

After the device is discovered successfully, use connectDevice to make a connection with the phone app.

```
    iHealthDevicesManager.getInstance().connectDevi
ce(userName, mac, type);
```

```
    private iHealthDevicesCallback
iHealthDevicesCallback = new
iHealthDevicesCallback() {
      @Override
          public void
onDeviceConnectionStateChange(String mac, String
deviceType, int status) {
      }
     };
```

To get the iHealth device controller in our case it's BG5 based on specs [4] we use the following:

```
Bg5Control bg5Control =
iHealthDevicesManager.getInstance().getBg5Control(m
ac);
```

3. iHealth device integration example

Before working with iHealth devices SDK you need to learn android multithreading communication pattern with Handlers and Messages. You must have complete understanding of those two classes and how they are used to communicate and pass data between two threads.

Therefore, how the SDK works – you need to register a callback (iHealthDevicesCallback) to receive connection state and perform operations on it. This callback will be triggered with startDiscovery() method within iHealthDevices singleton instance. This callback interface has several methods that needs to be overridden:

```
onScanDevice(String mac, String deviceType, int
rssi)
onScanFinish()
onDeviceConnectionStateChange(String mac, String
deviceType, int status, int errorID)
```

If any iHealth device is found onScanDevice() method will be called. To connect to that device send a command to the handler CONNECT_DEVICE

```
myHandler.sendEmptyMessage(CONNECT_DEVICE);
```

Those command codes needs to be declared first as constants.

```
private static final int ADD_SUCCESS = 101;
private static final int ADD_FAIL = 102;
private static final int SCAN_DEVICE = 103;
private static final int CONNECT_DEVICE = 104;
private static final int DISCONNECT_DEVICE = 105;
```

If a device is connected or disconnected or failed onDeviceConnectionStateChange() method will be called. Constants to determine the states are:

```
iHealthDevicesManager.DEVICE_STATE_CONNECTED
iHealthDevicesManager.DEVICE_STATE_CONNECTIONFAIL
iHealthDevicesManager.DEVICE_STATE_DISCONNECTED
```

In addition, others like these. Take control over your device if connection is successful. In our case it's BG5:

```
Bg5Control bg5Control =
iHealthDevicesManager.getInstance().getBp7Control(m
Address);
```

For different devices this control classes will be different. But naming conventions are the same.

Okay it was a brief overview. Lets follow some steps so that we can successfully connect our iHealth Device within out app.

After you put your binaries(.jar, .so) into /app/libs folder of your project, compile all of the files on build.gradle, Initialize iHealth SDK on your application class

```
iHealthDevicesManager.getInstance().init(this);
```

Create a Class named MeasureHelperIHealth that is responsible for connecting to the device.

Register a callback and add device filter (bg5 in our case)

```
public MeasureHelperIHealth(Context context) {
this.context = context;
```

```
callbackId =
iHealthDevicesManager.getInstance().registerClientC
allback(miHealthDevicesCallback);
iHealthDevicesManager.getInstance().addCallbackFilt
erForDeviceType(callbackId,
iHealthDevicesManager.TYPE_BG5);
iHealthDevicesManager.getInstance().sdkUserInAuthor
(context, userName, clientId,
clientSecret, callbackId);
myHandler = new MyHandler();
}
```

Remember to call sdkUserInAuthor() method to verify your identity. Client ID and Client Secret can be found registering iHealth website and adding new app there.

Use the iHealth Device Callback like it was described before.

Use myHandler to send message to the background thread in which we can call connectDevice(), startDiscovery(), disconnect() methods on background.

On onDeviceConnectionStateChange() method check if device is connected. If yes then open an activity or fragment of whatever.

Perform operation on that fragment/activity like measuring etc.

And at last - Trigger Device discovery for the first time so that callback is operational.

```
iHealthDevicesManager.getInstance().startDiscovery(
iHealthDevicesManager.DISCOVERY_BG5);
```

Now you can step further and work on the measurements itself, you can do it by yourself, depending on the device you have. Do not forget to unregister after you finish measurements Unregister on destroy:

```
iHealthDevicesManager.getInstance().unRegisterClien
tCallback(clientCallbackId);
```

2.5.4 Report requirements and tasks.

1. Learn the information about the iHealth SDK
2. Learn Android multithreading communication pattern with Handlers and Messages, links are in the literature, [1,2]
3. Successfully connect any device via Bluetooth or usb.

4. Try to write the measurements for any device.

2.5.5 Test questions.

1. Which methods can you use directly via API?
2. What is Handlers, in Android, how is it working?
3. What is Messages in Android, how is it working?

2.5.6 Literature

1. iHealthLabs. OpenAPI-V2-IOS. https://github.com/iHealthLabs/OpenAPI-V2-IOS_
2. iHealthLabs. OpenAPI-V2. https://github.com/iHealthLabs/OpenAPI-V2/
3. rimonxyz. ihdevicexamples. https://github.com/rimonxyz/ihdevicesexamples/tree/master/examples/Android_SDK
4. Wireless Smart Gluco-Monitoring System. https://cloud.c2m.net/ihealth/wireless-smart-gluco-monitoring-system

APPENDIX A. TEACHING PROGRAM OF THE MASTER COURSE “MOBILE AND HYBRID IoT COMPUTING”

DESCRIPTION OF THE MODULE

TITLE OF THE MODULE	Code
Mobile and hybrid IoT-based computing	**MC3**

Teacher(s)	Department
Coordinating: Dr. Butenko V.O. **Others: Dr. Odarushchenko O.M., Dr. Strjuk O.Y., Dr. Odarushchenko O.B., Butenko D.A.**	Computer Systems, Networks and Cybersecurity

Study cycle	Level of the module	Type of the module
MC	A	Full-time tuition

Form of delivery	Duration	Langage(s)
full-time tuition, distance tuition	Five weeks	English

Prerequisites	
Prerequisites: Need for training of developers creating software for connected devices or the Internet of Things	**Co-requisites (if necessary):**

Credits of the module	Total student workload	Contact hours	Individual work hours
7,5	230	148	82

Aim of the module (course unit): competences foreseeen by the study programme		
The aim of the module is to introduce the students to design and of mobile and IoT applications and services.		
Learning outcomes of module (course unit)	**Teaching/learning methods**	**Assessment methods**
At the end of course, the successful student will be able to: 1.Evaluate critical design tradeoffs for different mobile and IoT technologies, architectures,	Interactive lectures, Practicals	Module Evaluation Questionnaire

interfaces impact on usability, security, privacy of mobile and IoT computing services and applications; design, develop and publish their apps on different OS		
2. Perform the basic of cloud computing on various architectures, such as SaaS, PaaS, IaaS	Lectures, Practicals and seminars	Module Evaluation Questionnaire
3. Capture, analyze, search, share, store, process and intergrade big data for mobile applications	Lectures, Practicals and seminars	Module Evaluation Questionnaire

Themes	**Contact work hours**							**Time and tasks for individual work**	
	Lectures	Consultations	Seminars	Practiacl work	Laboratory work	Placements	**Total contact work**	**Individual work**	**Tasks**
1.Mobile and Networking	44			36			**80**	**38**	1.1 Introduction to the course: history of mobile, cloud and IoT development, basic standards, development guidelines. Developing applications for Android. Basic interaction types of IoT and Android applications.

									Developing applications for iOS. Basic interaction types of IoT and iOS applications. Usability, security and privacy concepts for Android and iOS apps. Basics of wearable programming Applications development for Android wearable. Applications development for iOS wearable.
2. Cloud Computing and IoT	20		6	12			**38**	**22**	2.1 Introduction to the Cloud Computing. Dynamic interactions and computing architectures – SaaS, PaaS, IaaS benefits, issues and concerns Economics of Cloud Computing. Service models, value

									and risks. Perform computing in Android applications on the cloud. Perform computing of iOS applications on the cloud.
3. Intregration of big data and IoT/IoE technologies	18		6	6			**30**	**22**	3.1 Integration of Big Data and IOT Technologies. Foundations for Big Data Systems for IoT. Big Data characteristics and tyeps. Big Data platform stack and tools. Architectures of Big Data systems. Requirements for Big Data systems
Total	**82**		**12**	**54**			**148**	**82**	**230**

Assessment strategy	**Weight in %**	**Dead lines**	**Assessment criteria**
Lecture activity, learning in laboratories	40	4	85% – 100% Outstanding work, showing a full grasp of all the questions answered. 70% – 84% Perfect or near perfect answers to a high proportion of the questions answered. There should be a thorough understanding and appreciation of the

			material. 60% – 69% A very good knowledge of much of the important material, possibly excellent in places, but with a limited account of some significant topics. 50% – 59% There should be a good grasp of several important topics, but with only a limited understanding or ability in places. There may be significant omissions. 45% – 49% Students will show some relevant knowledge of some of the issues involved, but with a good grasp of only a minority of the material. Some topics may be answered well, but others will be either omitted or incorrect. 40% – 44% There should be some work of some merit. There may be a few topics answered partly or there may be scattered or perfunctory knowledge across a larger range. 20% – 39% There should be substantial deficiencies, or no answers, across large parts of the topics set, but with a little relevant and correct material in places. 0% – 19% Very little or nothing that is correct and relevant.
Module Evaluation Quest	60	4	The score corresponds to the percentage of correct answers to the test questions

Author	**Year of issue**	**Title**	**No of periodic al or volume**	**Place of printing. Printing house or intrenet link**
Compulsory literature				
	2017	The Swift Programming Language (Swift 4.0.3)		Apple Inc., p. 500
	2017	Using Swift with Cocoa and Objective-C (Swift 4.0.3)		Apple Inc., p. 100

Matt Neuburg	2017	iOS 11 Programming Fundamentals with Swift		O'Reilly Media, October 2017, 646 P.
Marko Gargenta, Masumi Nakamura	2014	Learning Android, Develop Mobile Apps Using Java and Eclipse, 2nd Edition		O'Reilly Media, 2014, 286 p.
John Horton	2015	Learning Java by Building Android Games		John Horton, 2015, 392 P.
Justin Garrison, Kris Nova	2017	Cloud Native Infrastructure: pattern for Scalable Infrastructure and Applications in Dynamic Environment		O'Reilly Media, 160P.
Additional literature				
Fei Hu	2016	Security and Privacy in Internet of Things (IoTs) Models, Algorithms, and Implementations		2016 by Taylor & Francis Group, LLC CRC Press is an imprint of Taylor & Francis Group, an Informa business
Adrian McEwen, Hakim Cassimally	2014	Designing the Internet of Things		2014 John Wiley and Sons, Ltd.
	2016	Digitising the Industry Internet of Things Connecting the Physical, Digital and Virtual Worlds		River Publishers

АНОТАЦІЯ

УДК 004.382.74iOS _And:004.411](076.5)=111

Бутенко В.О., Одарущенко О.М., Стрюк О.Ю., Одарущенко Е.В. **Мобільні і гібридні обчислення на основі інтернету речей.** / За ред. Харченка В.С. – МОН України, Національний аерокосмічний університет ім. М. Є. Жуковського «ХАІ». – 124 с.

Викладено матеріали практичної частини курсу "MC3. Mobile and hybrid IoT-based computing", підготовленого в рамках проекту ERASMUS+ ALIOT "Internet of Things: Emerging Curriculum for Industry and Human Applications" (573818-EPP-1-2016-1-UK-EPPKA2-CBHE-JP).

Навчальний матеріал, представлений у цій практичній частині магістерського курсу, висвітлює основні теми розробки додатків для iOS та Android та використання їх для систем IoT.

Основні теми практичних робіт наступні:

- початок роботи з XCode та Android Studio - налаштування середовища розробки;
- дизайн та основні схеми діабетичної програми відстеження діагнозу «Глюкоза» для iOS та Android;
- переклад дизайну в код - додавання та налаштування основних компонентів «Глюкози»;
- початок роботи зі сховищами даних для iOS та Android;
- оцінювання інформації про стан здоров'я користувачів за допомогою HealthKit та Google Fit;
- Інтеграція сторонніх трекерів та глюкометрів за допомогою API.

Призначено для інженерів, розробників та науковців, які займаються розробкою та впровадженням IoT для промислових систем, для аспірантів університетів, які навчаються за напрямом комп'ютерних наук, комп'ютерної та програмної інженерії, а також для викладачів відповідних курсів.

Бібл. – 38, рисунків – 66.

ЗМІСТ

СКОРОЧЕННЯ 3
ВСТУП 4
1. РОЗРОБЛЕННЯ IOT ЗАСТОСУНКІВ ДЛЯ IOS 5
1.1. ПОЧАТОК РОБОТИ З XCODE – ВСТУП ДО IDE 5
1.2. ПРОЕКТУВАННЯ ТА БАЗОВІ СХЕМИ ЗАСТОСУНКА ДІАБЕТИЧНОГО ТРЕКЕРА НА IOS “GLUCOSE” 17
1.3. ПЕРЕТВОРЮВАННЯ ПРОЕКТУ В КОД – ДОДАВАННЯ ТА НАСТРОЮВАННЯ БАЗОВОГО КОМПОНЕНТУ ЗАСТОСУНКУ 23
1.4. ПОЧАТОК РОБОТИ З CORE DATA 41
1.5. ДОСТУП ДО ІНФОРМАЦІЇ ПРО ЗДОРОВ’Я КОРИСТУВАЧА ЗА ДОПОМОГОЮ HEALTHKIT 49
1.6. ІНТЕГРУВАННЯ СТОРОННІХ ПРИСТРОЇВ ЗА ДОПОМОГОЮ BLUETOOTH 61
2.РОЗРОБЛЕННЯ IOT ЗАСТОСУНКІВ ДЛЯ ANDROID 75
2.1. ПОЧАТОК РОБОТИ З ANDROID STUDIO – ВВЕДЕННЯ ДО ІНТЕГРОВАНОГО СЕРЕДОВИЩА РОЗРОБКИ 75
2.2 ПРОЕКТУВАННЯ ТА БАЗОВІ СХЕМИ ЗАСТОСУНКА ДІАБЕТИЧНОГО ТРЕКЕРА НА ANDROID “GLUCOSE” 82
2.3. ПЕРЕТВОРЮВАННЯ ПРОЕКТУ В КОД – ДОДАВАННЯ ТА НАСТРОЮВАННЯ БАЗОВИХ ЕЛЕМЕНТІВ ЗАСТОСУНКУ “GLUCOSE” 89
2.4. ПОЧАТОК РОБОТИ З БАЗАМИ ДАНИХ НА ANDROID 101
2.5. ІНТЕГРУВАННЯ СТОРОННІХ ТРЕКЕРІВ ТА ГЛЮКОМЕТРІВ ЗА ДОПОМОГОЮ ПРИКЛАДНОГО ПРОГРАМНОГО ІНТЕРФЕЙСУ 109
ДОДАТОК А. НАВЧАЛЬНА ПРОГРАМА МАГІСТЕРСЬКОГО КУРСУ “MOBILE AND HYBRID IOT COMPUTING” 116
АНОТАЦІЯ ТА ЗМІСТ 123

CONTENTS

ABBREVIATIONS	3
INTRODUCTION	4
1. DEVELOPING IOT-BASED APPLICATIONS FOR IOS	5
1.1. GETTING STARTED WITH XCODE – INTRODUCTION TO THE IDE	5
1.2. DESIGN AND BASIC LAYOUTS OF THE IOS DIABETIC TRACER APPLICATION “GLUCOSE”	17
1.3. TRANSLATING DESIGN INTO CODE - ADD AND SETUP BASIC APPLICATION COMPONENT	23
1.4. GETTING STARTED WITH CORE DATA	41
1.5. ACCESSING USER HEALTH INFORMATION USING HEALTHKIT	49
1.6. INTEGRATING THIRD-PARTY DEVICES THROUGH BLUETOOTH	61
2. DEVELOPING IOT-BASED APPLICATIONS FOR ANDROID	75
2.1. GETTING STARTED WITH ANDROID STUDIO – INTRO TO THE DEVELOPMENT ENVIRONMENT	75
2.2 DESIGN AND BASIC LAYOUTS OF THE ANDROID DIABETIC TRACKER APPLICATION “GLUCOSE”	82
2.3. TRANSLATING DESIGN INTO CODE – ADD AND SETUP BASIC “GLUCOSE” ELEMENTS	89
2.4. GETTING STARTED WITH DATABASES ON ANDROID	101
2.5. ACCESSING USER HEALTH INFORMATION USING GOOGLE FIT	109
2.6. INTEGRATING THIRD-PARTY TRACKERS AND GLUCOMETERS USING API	116
APPENDIX A. TEACHING PROGRAM OF THE MASTER COURSE “MOBILE AND HYBRID IoT COMPUTING”	
ABSTRACT AND CONTENTS	136

Бутенко Валентина Олегівна
Одарущенко Олег Миколайович
Стрюк Олексій Юрійович
Одарущенко Олена Борисівна
Бутенко Дмитро Анатолійович

МОБІЛЬНІ І ГІБРИДНІ ОБЧИСЛЕННЯ НА ОСНОВІ ІНТЕРНЕТУ РЕЧЕЙ

Практикум
(англійською мовою)

Редактор Харченко В.С.

Комп'ютерна верстка
В.С. Харченко,
О.О. Ілященко

Зв. план, 2019
Підписаний до друку 20.02.2017
Формат 60х84 1/16. Папір офс. No2. Офс. друк.
Умов. друк. арк. 7,27. Уч.-вид. л. 7,81. Наклад 150 прим.
Замовлення 220819-4.

Національний аерокосмічний університет ім. М. Є. Жуковського
"Харківський авіаційний інститут"
61070, Харків-70, вул. Чкалова, 17
http://www.khai.edu

Випускаючий редактор: ФОП Голембовська О.О.
03049, Київ, Повітрофлотський пр-кт, б. 3, к. 32.
Свідоцтво про внесення суб'єкта видавничої справи до державного реєстру видавців,
виготовлювачів і розповсюджувачів видавничої продукції
серія ДК No 5120 від 08.06.2016 р.

Видавець: ТОВ «Видавництво Юстон»
01034, м. Київ, вул.. О. Гончара, 36-а, тел.: +38 044 360 22 66
www.yuston.com.ua
Свідоцтво про внесення суб'єкта видавничої справи до державного реєстру видавців,
виготовлювачів і розповсюджувачів видавничої продукції
серія ДК No 497 від 09.09.2015 р.