Oskar Gusgård

Application development for the Apple Watch

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology

Thesis

14 May 2018

Metropolia

Author Title Number of Pages Date	Oskar Gusgård Application development for the Apple Watch 36 pages + 2 appendices 14 May 2018
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Software Engineering
Instructors	Peter Hjort, Senior Lecturer

The thesis was conducted as a project for a Finnish startup company, with the goal of creating a supporting Apple Watch application for an iOS map and voice guidance-based route application. The Apple Watch application's goal was to make the Route Pepper application more user-friendly during exercise.

For the project to succeed, the most important features of the iOS application had to be created for the Apple Watch application, utilizing the watchOS's WatchKit framework, the Core Location framework and the Watch Connectivity framework for establishing the connection between the applications; all while avoiding the limitation of the watchOS.

The start of the project focused on studying the Apple Watch technology, architecture and its features. Studying the communication establishment between the iOS and the watchOS application was one of the key subjects for creating a Watch application. After the study, the connection was established for sharing data between the applications and to send Pep Point coordinates for the iOS application to be processed. A notification system was created for sending map images generated by the iOS application to be shown to the user upon entering a Pep Point location. The images contain the route line and a marker for the Pep Point coordinate.

During the project, several restrictions were encountered with the watchOS operating system, which hindered the creation of some of the planned features. Problems were faced with the WatchKit framework's WKInterfaceMap class' limitations, which meant that a map interface could not be created. A compromise was made by generating map images on the iOS application which were sent to the Watch application. WatchKit did not provide support for the Firebase API, which meant that all Firebase functionality had to be executed on the iOS application.

Project results were found satisfying and the created Watch application was published in the Apple App Store to support the iOS application. The project was a great learning experience for all parties involved.

Keywords	wearable technology, location-based applications, iOS, watchOS

Metropolia

Tekijä Otsikko Sivumäärä Aika	Oskar Gusgård Sovelluskehitys Apple Watchille 36 sivua + 2 liitettä 14.6.2018
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Software Engineering
Ohjaaja	Lehtori Peter Hjort

Insinöörityö toteutettiin projektina suomalaiselle start-up-yritykselle, ja tavoitteena oli luoda kehityksen alla olleelle kartta- ja ääniohjeistuspohjaiselle reittiopastus-iOS-sovellukselle sitä tukeva Apple Watch -sovellus. Toteutettavan sovelluksen tavoitteena oli tehdä sovelluksesta käyttäjäystävällisempi liikunnan yhteydessä.

Projektin onnistumiseksi sovellukseen tuli luoda tärkeimmät iOS-sovelluksen toiminnollisuuksista käyttäen watchOS:n WatchKit-kehystä, Core Location -kehystä ja Watch Connectivity -kehystä sovellusten keskinäisen viestinnän toteuttamiseksi. Apple Watchin rajoitteita tuli kiertää toiminnollisuuksien toteuttamiseksi.

Projektin alussa perehdyttiin Apple Watchin teknologiaan, arkkitehtuuriin ja tärkeimpiin ominaisuuksiin. iOS- ja watchOS-sovellusten yhteiseen kommunikaatioon tutustuminen oli yksi tärkeimmistä osa-alueista Watch-sovelluksen luonnissa. Tutustumisen jälkeen sovellusten välille luotiin kommunikaatioyhteys sovellusten yhteisen datan jakamiselle ja Pep Point -koordinaattipisteiden lähettämiselle iOS-sovelluksen käsiteltäväksi. Sovellusten välille luotiin ilmoitusjärjestelmä, jonka avulla iOS-sovellukselta lähetettiin reittiviivan ja Pep Point -koordinaatinpisteen sisältämiä karttakuvia Watch-sovellukselle käyttäjälle näytettäväksi.

Projektin aikana watchOS-käyttöjärjestelmässä havaittiin useita rajoituksia, jotka vaikeuttivat toivottujen ominaisuuksien luontia. Ongelmia aiheutti WatchKit-kehyksen WKInterfaceMap-luokan rajallisuus, jonka takia toivottua karttanäkymää ei voitu luoda. Tämä kierrettiin luomalla karttakuvia iOS-sovelluksella ja lähettämällä ne Watch-sovellukselle. WatchKit ei myöskään tarjonnut tukea Firebase-ohjelmointirajapinnalle, minkä seurauksena kaikki Firebase-toiminnallisuus suoritettiin iOS-sovelluksessa.

Projektin tuloksiin oltiin tyytyväisiä, ja toteutettu Watch-sovellus julkaistiin Apple-kauppaan iOS-sovelluksen tueksi. Projekti oli erittäin opettavainen niin tilaajalle kuin työn toteuttajalle.

Avainsanat	puettava teknologia, paikkatietosovellukset, iOS, watchOS

Metropolia

Contents

List of Abbreviations

1 Introduction 1

2 Route Pepper 1

3 Apple Watch and watchOS 4

3.1 Apple Watch interaction 4
3.2 Notifications 5
3.2.1 Notification appearance rules 6
3.2.2 Apple Watch notifications 6
3.3 Complications 8

4 Watch application architecture 9

4.1 iOS application 10
4.2 Watch App 11
4.3 WatchKit Extension 11
4.4 WatchKit 11
4.4.1 Application states 11
4.4.2 Background tasks 13
4.4.3 Snapshots 14
4.5 Watch Connectivity 14
4.5.1 WCSession 14
4.5.2 Sending data 15
4.5.3 Receiving data 16

5 The project 17

5.1 Development tools 18
5.1.1 Swift 18
5.1.2 Xcode 18
5.1.3 Google Firebase 20
5.2 Communication model 22
5.3 The application 23
5.3.1 Pep Point notifications 25
5.3.2 Adding a Pep Point 28
5.4 Sharing data between the applications 31

Metropolia

5.5 Testing 31
5.6 The encountered problems 32
5.7 Further development 35
6 Conclusion 35
References 37
Appendices
Appendix 1. MapSnapshotManager class
Appendix 2. Add Pep Point method on Watch Application

Metropolia

List of Abbreviations

iOS	Apple iPhone's operating system.
watchOS	Apple Watch's operating system.
macOS	Apple Mac's operating system.
tvOS	Apple Tv's operating system.
LTE	Long-Term Evolution. Telecommunications standard for high-speed wireless communication for mobile devices.
APN	Apple Push Notification service.
WCSession	Watch Connectivity session for establishing two-way communication between the devices.
WKExtension	Apple Watch extension for performing app-level tasks.
IDE	Integrated development environment.
UIKit	Framework for constructing user interfaces for the iOS and tvOS.
GPS	Global positioning system.
GPX	GPS exchange format. XML format for exchanging GPS data.
HTTP	Hypertext transfer protocol.
MVP	Minimum viable product.

1 Introduction

As technology becomes smaller and smarter, it has become possible to actually wear it. One of today's biggest tech trends is wearable technology and its market is predicted to double by the end of 2021, the most popular categories being smart watches and wristbands [1].

This Bachelor's Thesis explores wearable technology through the study of Apple Watch and watchOS development. The thesis project was conducted for a Finnish start-up company Appsipaja Oy, with the goal of developing an Apple Watch application for Route Pepper, which is a map and voice guidance application for iOS.

The requirements for the project was to create an Apple Watch application to serve as an extension to the iOS application, with the convenience of using the Route Pepper application directly from the watch while exercising. From the Watch application, the user had to be able to change the application settings, add Pep Point coordinates to the current location, see the trail on a map and to be able to start routes directly from the watch. Some of the goals could not be reached during the project timespan, but compromises where made to reach a satisfying product.

The thesis covers basics of Apple Watch technology, Watch application architecture and hierarchy, communication between the iOS and Watch applications, the development tools and the project itself.

2 Route Pepper

Route Pepper is a map and guidance application designed for outdoor activities, mainly running, walking and cycling, but can be used for other activities that could require location guidance. Route Pepper tackles a problem that many have when designing new routes and try them for their first time: the worry of going off route and the constant glances on the phone's map application.

Route Pepper offers a solution: an application which will guide the user on their route by providing notification and audio guidance. The routes are created online via Route

Pepper Planner -tool (see Figure 1). The tool allows the creation of routes with a trail path and Pep Point markers. Pep Points are markers that contain a user provided message, which will be read to the user upon reaching the coordinate location. Figure 1 displays the information a Pep Point contains.

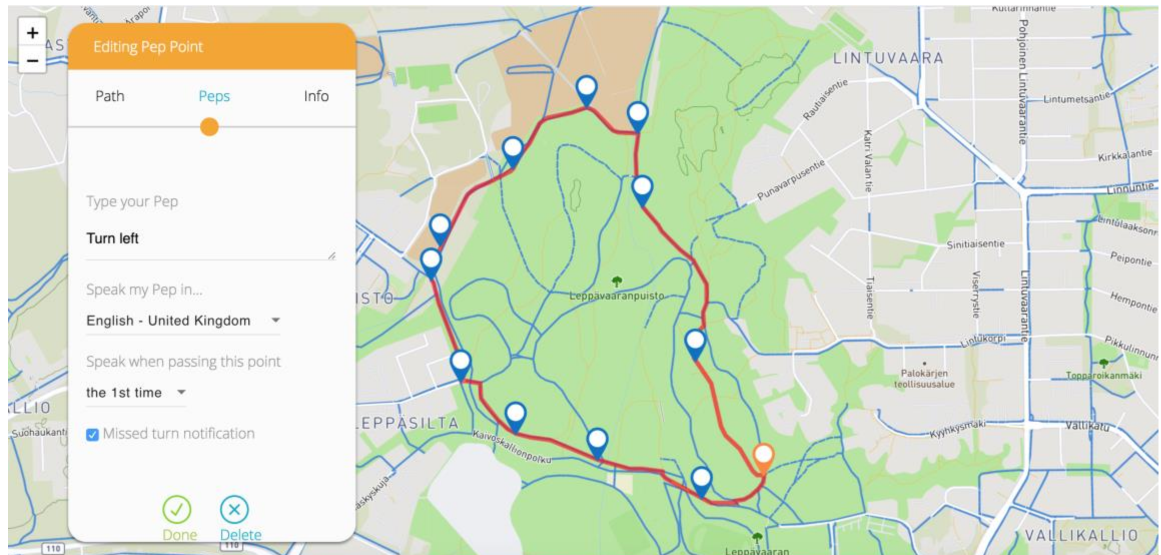

Figure 1. Route Pepper Planner web tool for creating routes.

The newly created route can be seen on the Route Pepper iOS application’s route list. After the user has selected a route, the route can be started, and the user can care freely test out the route, while keeping their phone in the pocket. Figure 2 displays the route screen after the it has been selected from a list of all the available routes.

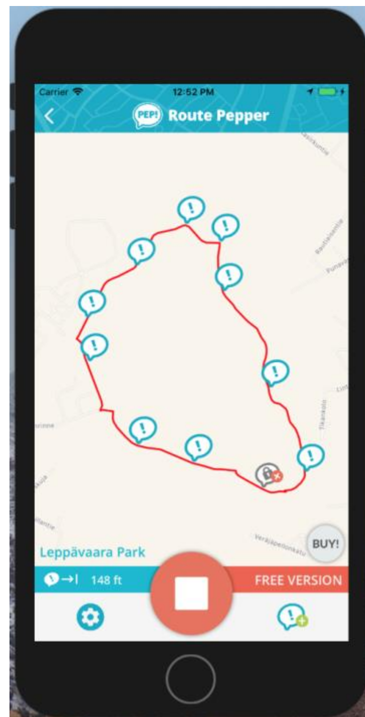

Figure 2. iOS application’s route screen.

Route Pepper can be used for much more than to discover new routes. The application could be used for marathon training or similar sports events. Remembering long routes can be quite tough, and the user could also place Pep Points along the route to remind about hydration and supplements. Route Pepper could also be used for hikers or sightseeing to notify of points of interest.

The application also features a tool for route sharing. User can import shared routes to get instant access to a new route. The user can export routes to make the routes public for the rest of the community.

Due to the rising popularity of wearable technology, Route Pepper has also been developed for the Apple Watch. Creating a Watch application makes the application more convenient for the user. The routes can be launched directly from the watch without the need of interacting with the phone, and the Pep Point notification will be displayed on the watch, containing a map image of the Pep Point location. New points can conveniently be created with the Watch application while exercising.

3 Apple Watch and watchOS

The original Apple Watch was released April 24th of 2015, becoming the best-selling wearable of the year with 4,2 million devices sold during the 2nd quarter of 2015; beating its Android competitors such as Fitbit and Xiaomi [2].

Since the original release of the Apple Watch, the device has rapidly been improved with updated versions. Apple Watch Series 1 and Series 2 were both released during 2016 and Series 3 in 2017. Each one of the Series featured updates to the previous model, such as upgraded processing power, improved memory and storage and Bluetooth connectivity. The Series 3 also featured a model with its own LTE cellular network, which gives the opportunity to use the watch applications without using the phone’s network. [3.]

The watchOS was released along with the original Apple Watch in 2015 and it is based on the iOS operating system; having many similar features [4]. As new versions of the watchOS were released, more access for the developers was granted; watchOS 2 featured support for native third-party apps [5]. The newest version of the operating system, watchOS 4, was released along with the Series 3 in September 19th of 2017 [6].

3.1 Apple Watch interaction

Due to the small size of the device, the interaction is much more limited compared to the iOS. The gestures are very limited, only supporting vertical and horizontal swipes and taps. The Apple Watch does not support multi-touch gestures. [7, p. 62.] To make up for the limited on-screen interaction, the watch introduces the force touch and the digital crown.

The watch has tiny electrodes on the display to determine if a tap is light or if it has more pressure applied to it. A force touch is triggered by forcefully pressing the screen which will open a context menu (see Figure 3). Context menus are not mandatory, but they are featured in most applications.

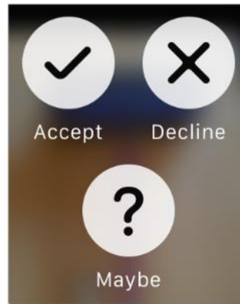

Figure 3. Force touch menu on the watch [8].

The digital crown lets the user scroll the screen or its elements without obstructing the screen. The watch screen can also be scrolled with a finger gesture, just as on the iOS, but most of the time the user's finger would hide the element being scrolled and the user would lose track of how far to scroll. The digital crown allows the user to scroll elements without any obstruction. The digital crown can be pressed down to exit the application and access the application menu.

3.2 Notifications

Notifications are mainly used to communicate with the user while the application is not in the foreground. Scheduled notifications are provided with a specific set of content and are triggered by a time or location value. Notifications can contain text, images, video or audio. Notifications mostly function the same way on watchOS, as they do on the iOS but comes with a set of new scheduling and display rules. Notifications can either be scheduled remotely from a server or locally on the device. Local notifications are scheduled and handled using the User Notifications framework. [9.]

Remote notifications can be scheduled using the Apple Push Notification service (APN). The remote notifications can be established by setting up an automatic IP connection between the application and the APN, and between a server and the APN [10]. Remote notifications are supported by iOS, tvOS, watchOS and macOS [9].

Local notifications are commonly used for alerting the user of new data the application has downloaded in the background from a server. The local notifications are also useful for calendar, to-do list and map applications that trigger when the user reaches a certain

location. Local notifications can be scheduled from either the iPhone or the Apple Watch [9].

3.2.1 Notification appearance rules

Pairing an Apple Watch with an iPhone significantly changes the way notifications are displayed for the user (see Figure 4). Remote and local (scheduled from the iOS application) notification will be displayed either on the iOS or the watchOS device depending on which one is being actively used. If the iPhone is in an unlocked state when the scheduled notification would appear, it will be displayed on the iPhone. If the iPhone is in a locked state, the notification will automatically be displayed on the Apple Watch. The notification will be displayed on the locked iPhone if the Apple Watch is not in a reachable mode, meaning that it cannot be reached via Bluetooth or Wi-Fi. Local notifications scheduled on the Apple Watch will only be displayed on the watch.

Notification Type	Source	Destination
Local	iOS app	Either Apple Watch or iPhone, depending on the locked/unlocked state of both devices.
Local	WatchKit extension	Apple Watch only
Remote	Your server	Either Apple Watch or iPhone, depending on the locked/unlocked state of both devices.
Silent	Your server	iPhone only

Figure 4. Notification rules when an iPhone is paired with an Apple Watch [11].

3.2.2 Apple Watch notifications

The Apple Watch has a unique way of displaying notifications. When a notification first arrives to the watch, the device will slightly vibrate and emit an alert sound. When the user raises their wrist to look at the notification, the watch displays a short-look view of the notification. [11.]

The short-look is a view that only displays a very bare version of the notification (see Figure 5). The short-look is designed to give the user a quick glimpse of the notification’s contents. The look displays the application icon along with the title and the application name. Short-looks are automatically generated by the system and the elements cannot

be customized by the developer. The notification title string can be customized when the notification is being created.

Figure 5. Short-look view of a watch notification [11].

If the user continues to look at the notification after the short-look interface has been displayed, the interface will quickly transition into a long-look view (see Figure 6). Long-look views are scrollable interfaces that display the notification content and any action buttons configured for it. Long-looks are divided into three different parts: the sash, the content area and the bottom area.

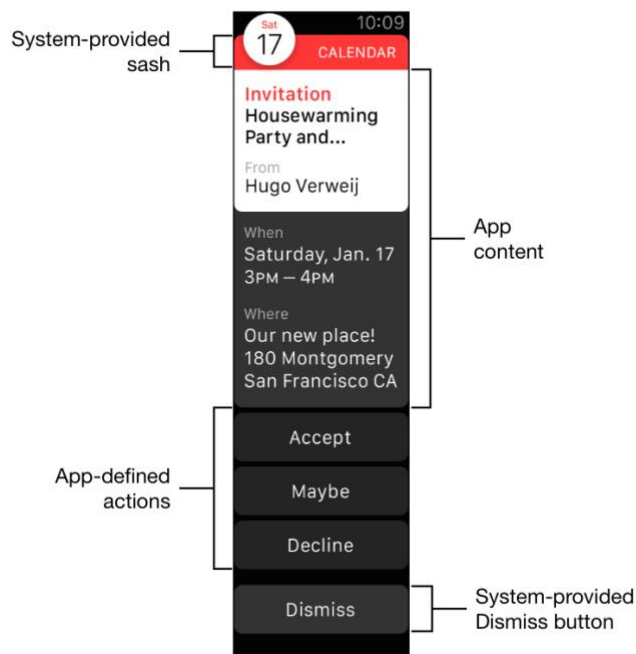

Figure 6. Long-look view of a Watch application’s notification [11].

The sash is an overlay that contains the application's name and logo. The content area displays the information attached to the notification, such as images and text. The bottom area contains any action buttons that have been configured for the notification type. [11.]

Pressing one of the action buttons will deliver the action either to the iOS application or the Watch application. Foreground actions will always be delivered to the Watch application. Background actions for local notifications will be delivered to the Watch application, whereas background actions for remote notifications will be delivered to the iOS application. The dismiss button is provided by the system and cannot be customized; it will always be included in every notification. [11.]

Custom long-look interfaces consist of two interfaces: static and dynamic. Dynamic interfaces are highly customizable while the static is not. A long-look view will fall back into the static interface if the customization takes too long to process or when there has been an error. Static interfaces are predefined during design and will only display the sash, an alert message (containing the notification message string) and the dismiss button. [11.]

3.3 Complications

Compilations are small elements on the watch face (the main screen of the Apple Watch) that provide quick access to information provided by the corresponding application. Complications are created with the ClockKit framework. Complications can be designed for an application and they can be attached to the watch face by the user.

At minimum, complications show the application icon, but they can also display data generated by the application. A calendar application could show the current date, and an activity tracker the status of the day's goal. Tapping the complication launches the application. Complications feature a variety of different templates called the complication family (see Figure 7). There are six different types of complications: circular, modular small/large, utility small/large and extra-large. The complication templates can display either text, an image or both. [12.]

Figure 7. Watch complications and different complication families [7, p. 300].

Applications are not required to provide complications, but Apple highly recommends having at least one to act as a launcher for the application. There are several benefits for having the application's complication on the user's watch face: the application can conveniently show useful data to the user without starting the application, the application will be kept suspended in memory which lets the system rapidly wake it up when the complication is tapped, and it also gives the application a bigger budget for background tasks. [12.]

Because the user's interaction with the watch face occurs quick and does not last for long, the ClockKit framework must process the complications in advance so that they can always display the newest information to the user. When complication data is needed by the ClockKit, it runs the application's WatchKit extension and calls the methods for the complication data source object. The object conforms to the CLKComplicationDataSource protocol, which provides the background task methods for updating the complications. [12.] Due to power usage restrictions, the Apple Watch only allows 50 complication data pushed per day [7, p. 297].

4 Watch application architecture

A Watch application cannot be installed on the Apple Watch on its own; it always has to have a paired iPhone with the application installed on it. For the Watch application to function, the watch has to be connected to the paired iPhone via Bluetooth or Wi-Fi. The communication and the connection session are established with the Watch Connectivity

framework. The watch architecture consists of three main elements: iOS application, Watch application and the WatchKit Extension (see Figure 8) [13].

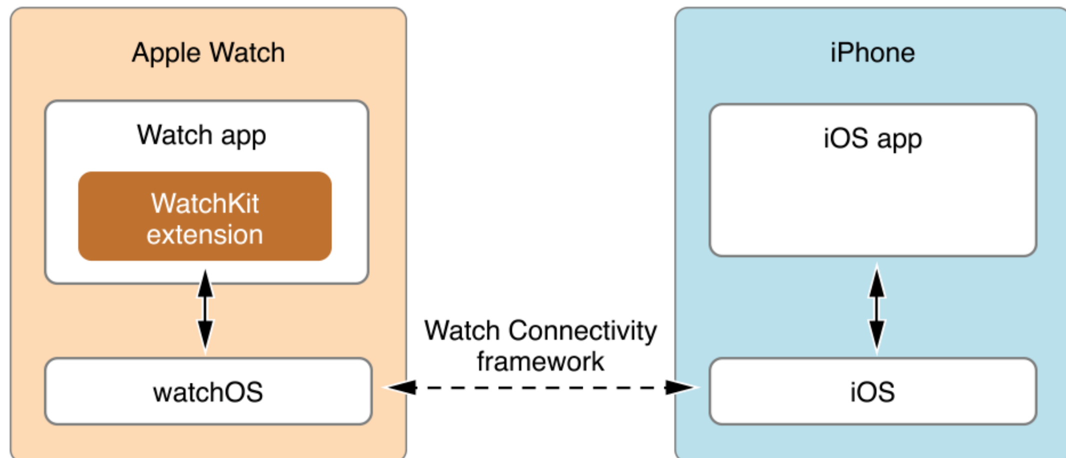

Figure 8. The Watch application architecture [13].

4.1 iOS application

The application installed on the user's iPhone is called the parent application. iOS application installed on the iPhone that has a Watch application, will automatically be installed on the Apple Watch as well. The Watch application installed on the Apple Watch is completely dependent on the iOS application. Some processes of the Watch application are often delegated over to the parent application, and sometimes the application only acts as a display for the data that has been generated over at the parent application. Some processes have to be delegated over to the parent applications since the watchOS does not have all of the same frameworks as iOS. During the project, one of the frameworks that was lacking from the watchOS was Firebase support, which meant that Firebase related work had to be conducted on the iOS application instead.

The parent application shares data with the Watch application through the WatchKit Extension. By creating a WCSession utilizing the Watch Connectivity framework, the applications can create a connection for transferring data such as user settings and application states.

4.2 Watch App

The Watch App is a bundle of resources that resides on the parent application and are installed on the Apple Watch. These resources include the application storyboard, image and audio files and any localization files used for the Watch application. Within the Watch App bundle resides the WatchKit Extension. [7, p.65.]

4.3 WatchKit Extension

The WatchKit Extension is the area of the application hierarchy that connects the Watch and the parent applications together. The extension contains the code written for the Watch application. The extension uses the WatchKit framework to manipulate the interface of a Watch application: it uses the classes of the framework to configure the Watch App's elements and create responses to the user interactions.

4.4 WatchKit

WatchKit is the framework for third-party developers for creating applications for the watchOS. WatchKit is watchOS's equivalent to the iOS's UIKit framework, as they both are used to create interfaces and interaction for the application [14]. A Watch application includes one or more interfaces which contain buttons, sliders, tables and many other visual elements. WatchKit uses the classes of the framework to configure these elements and to establish connection to the user interactions. The WatchKit framework provides a WKExtensionDelegate protocol which is a collection of methods that can be implemented to manage app-level behavior of the WatchKit Extension. The protocol mainly handles application states, snapshots and background tasks. [15.]

4.4.1 Application states

WatchKit notifies changes in the application's execution state to the extension delegate object. The state changes are triggered by major events in the lifetime of the application. Table 1 describes the different application states.

State	Description
Not running	The application has not been launched or has been terminated
Inactive	The application is running in the foreground but will not receive events. This state will occur before the application enters background or the device receives an interruption such as a phone call or a message
Active	The application is in the foreground and will receive events
Background	The application has been given background execution time to fetch updates
Suspended	The application is in memory but is not executing any code. The system suspends applications that are in the background which do not have any pending background processes

Table 1. Watch application states [16].

The WKExtensionDelegate can respond to state transition events by implementing the delegate's application lifecycle methods. These methods can, for example, be used to preload resources, configure initial user interfaces and save application states and user data. [16.] Figure 9 displays the flow between the application states and the lifecycle methods.

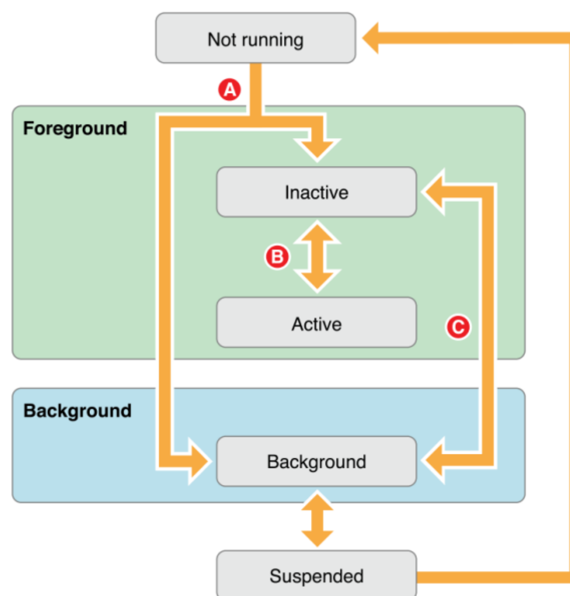

Figure 9. Interaction between different states of a Watch application [16].

The WatchKit calls the applicationDidFinishLaunching(_:) method after finishing to launch the application, but before the application's interface is visible (See A of Figure 9). The method can be used to setup configurations and to initialize which view controller will be shown to the user at launch. [17.] The applicationDidBecomeActive() method tells the delegate that the Watch application is now visible and the processes can be continued, whereas the applicationWillResignActive() will signal the opposite (See B of Figure 9) [18]. These methods are for pausing and resuming tasks.

The applicationWillEnterBackground() method will tell the delegate that the application is about to enter the background. The method can be used to store the application state and information to ensure that the right state is loaded after the application re-enters the foreground (see C of Figure 9). [19.] The applicationWillEnterForeground() method is called when the application is re-entering foreground after being backgrounded (see C of Figure 9). The method is used to reload the state which the applicationWillEnterBackgroud() has stored. [20.]

4.4.2 Background tasks

Background tasks give the application a small amount of time to run in the background. Several different handler types are provided for the background tasks, each for a specific type of activity. Some of these tasks can be manually scheduled by the application, while some are automatically scheduled by the system. [21.] Each background task is processed by the delegate's handle(_:) method, which determines the type of the task.

The user interface can be updated to a snapshot which has been received by the WKSnapshotRefreshBackgroundTask. When the application receives data from the iOS application via the Watch Connectivity framework, it is handled by the WKWatchConnectivityRefreshBackgroundTask. Background NSURLSession transfers are handled by the WKUrlSessionRefreshBackgroundTask. The task is triggered when a background transfer requires authorization or after a background transfer is completed. [21.]

4.4.3 Snapshots

Snapshots are glimpses of the application state that are displayed in the dock when the watch's side button is pressed. The system keeps the most recently used apps in the dock so that they can be resumed quickly. These apps receive priority for background tasks. The dock can hold up to 10 applications at the same time. [21.] Pressing the watch's side button behaves similarly as double-pressing the home button on an iPhone. Using the methods of the WKSnapshotRefreshBackgroundTask class, the snapshot can be updated, and updates can be manually scheduled. Snapshots will also be scheduled by the watchOS automatically in certain application states such as when the app enters foreground or background. [21.]

4.5 Watch Connectivity

Watch Connectivity is a framework for establishing a two-way communication between the applications. The communication is used for data sharing. [22.] The shared data can be just small pieces of data, such as dictionaries, or entire files. The framework is an essential part for making a Watch application. The framework can also be used to update the watch's complications. The data can be sent instantly or as background transfers.

4.5.1 WCSession

To initiate communication between the applications, both the Watch and the iOS applications have to establish communication using the WCSession class. Both applications have to setup their own sessions at some point of their execution. Listing 1 displays the methods required to establish the connection. To configure the session, a delegate has to be assigned to the default WCSession object, after which the session's activate() method has to be called to make the connection. [23].

```
if WCSession.isSupported(){
      let session = WCSession.default
      session.delegate = self
      session.activate()
}
```

Listing 1. Establishing a two-way connection with the counterpart application.

The isSupported() method can be used to determine whether the paired iOS device supports the Watch Connectivity framework; only devices with iOS version 9.0+ are supported [23]. Default session object is used for the activation. After the device has established its part of the connection, the device can use data transfer methods. Only background transfer methods can be used if the other device cannot be reached at the time.

Before transferring data, it is recommended to check whether the data can reach its destination. Session's isPaired() method is used to check whether the Apple Watch is paired to an iPhone and vice versa. On the iPhone, the isWatchAppInstalled() method can be called to check whether the Watch App has been installed on the paired device. For instant messaging, the session's isReachable() method must return true, or else the messages cannot reach the counterpart device.

4.5.2 Sending data

Most of the time, data is sent to the counterpart device as a dictionary, most commonly in the form of [String:Any]; an exception to this being the transfer of files which include a URL path to the file. Many of the data transfer methods send the data in the background, but the data can also be sent instantly if necessary.

To send regular dictionaries in the background, transferUserInfo(_:) and updateApplicationContext(_:) are used. Each dictionary sent by the former one is queued in the sending order to ensure that all of them are handled by the counterpart application [24]. The dictionaries sent by the latter, however, will replace themselves if a new one is sent [25]. Listing 2 displays the use of updateApplicationContext(_:); transferUserInfo(_:) is used in the same manner.

```
if WCSession.isSupported(){
      let session = WCSession.default
      if session.isWatchAppInstalled {
             let dictionary : [String : Any] = ["key":value]
             do {
              try session.updateApplicationContext(dictionary)
             } catch {
              print("Error: \(error)")
             }
      }
}
```

Listing 2. Sending a dictionary as a background transfer using updateApplicationContext(_:).

TransferUserInfo(_:) method is useful when sending data that has to be received regardless of any new data that will come afterwards. The method would be useful for a chat application whereas updateApplicationContext(_:) would be more suitable for sending application settings, since only the latest ones are needed. Complication data can be updated with transferCurrentComplicationUserInfo(_:).

Background file transfer can be done with the transferFile(_:metadata:) method. The method requires a reference to the file URL to identify the file. The file has to be local to the sending device to be reachable. The meta data can hold a dictionary for additional information on the file. [26.]

Instant data transfers can only be created using the sendMessage(_:replyHandler:errorHandler:) method. Listing 3 displays the use of the method. The sent dictionary will reach the counterpart application as soon as possible and return a dictionary as a reply. Calling this method from the Watch application will wake up the iOS device and make it reachable. Calling the method from the iOS application will not have the same effect. If the counterpart application is unreachable, the errorHandler will be executed. [27.]

```
let session = WCSession.default
session.sendMessage(["key" : value], replyHandler:
      { (dictionary : [String : Any]) in

            //The reply was received, handle the received dictionary of data

      }, errorHandler: { error in
            //Handle the error
})
```

Listing 3. Sending an instant message from the watch to the iOS application.

4.5.3 Receiving data

When the Watch application receives background data from the parent application, it is not guaranteed that it will wake up and handle the incoming data right away. Most of the time, the application will postpone the data handling to preserve battery usage. On watchOS 4 the application will immediately handle the data if it is the frontmost application and within 10 minutes for all other applications. For older watchOS’s, there is no guarantee to when the data will be processed; sometimes it will not be processed till the application is launched. [28.]

To receive data from the counterpart application, the application has to have a class that conforms to the WCSessionDelegate protocol. The delegate object has to be configured for the WCSession object at the time of making the connection. To handle incoming data, the class has to implement a ‘didReceive’ function for each method of sending data. Each of the data transferring methods have their own receiver method. Listing 4 displays the ‘didReceive’ function of the updateApplicationContext(_:) background transfer method. Only the sendMessage(_:replyHandler:errorHandler:) can send a reply to the sender application.

If an application uses the same method of sending data in different use cases, the sent data has to be made identifiable by the receiving application, since each of them will be received by the same method. As Listing 4 illustrates, a good way of identification is to place a descriptive string as the first key of the dictionary, and have it refer to the actual dictionary of data [“descriptiveTag” : [“key” : value, “key” : value]].

```
func session(_ session: WCSession, didReceiveApplicationContext
 applicationContext : [String:Any]) {
      //A good way of distinguishing different data is to have a descriptive
      key for identification
      if(applicationContext.keys.first == "description"){
            //Handle received data
      } else if … {
            //Handle received data
      }
…
}
```

Listing 4. Receiving a dictionary sent by updateApplicationContext(_:) method.

5 The project

The scope of the project was to create a Watch application for the Route Pepper application. The application serves as an extension to the iOS application which offers the user a more convenient way of using the application; the user can easily interact with the application while exercising. The project details will focus on the Watch application but will mention the iOS application when appropriate. Some of the feature goals could not be reached during the time allocated for the project; those features are discussed in this chapter’s Future Development section.

5.1 Development tools

Several development tools were utilized to accomplish the project goals. The project was developed using Apple's Xcode IDE, which is designed for application development for Apple's devices. Swift was used as the project's programming language, since it was used for the iOS application. Route Pepper is a serverless application built using the Google Firebase service. The Watch application indirectly connects itself to Firebase via the parent application.

5.1.1 Swift

Since the early days of Apple Development, Objective-C programming language has been used as the main language for most of Apple's products. The language was based on the C-programming language and was first used for the NeXTSTEP operating system released in 1988. [29]. June of 2014, at Apple's WWDC (Worldwide Developers Conference), Apple announced a new modernized programming language for their products: The Swift programming language [30].

Swift is an opensource programming language that was created to replace Objective-C for Apple's products. Both of the languages can still be coded side-by-side, which helps the transition to the new language [31]. Swift was designed to be faster, safer, easier to read and maintain [32]. Objective-C can still be used to code Apple products and is supported by the Xcode IDE.

5.1.2 Xcode

Xcode is an Apple developed IDE for macOS released in 2003. Xcode is designed for application development for Apple's operating systems but includes a wide support for multiple different programming languages.

Application scenes are designed in Xcode's storyboard window. The storyboards were first introduced with the iOS 5 [33]. Each application has their own storyboard file which describes the layout of the application. When creating a Watch application, both of the applications have their own storyboard files. The storyboard is a visual representation of

the application's scenes where new scenes and connections can be easily made without writing any code. Figure 10 displays an example of an application's storyboard.

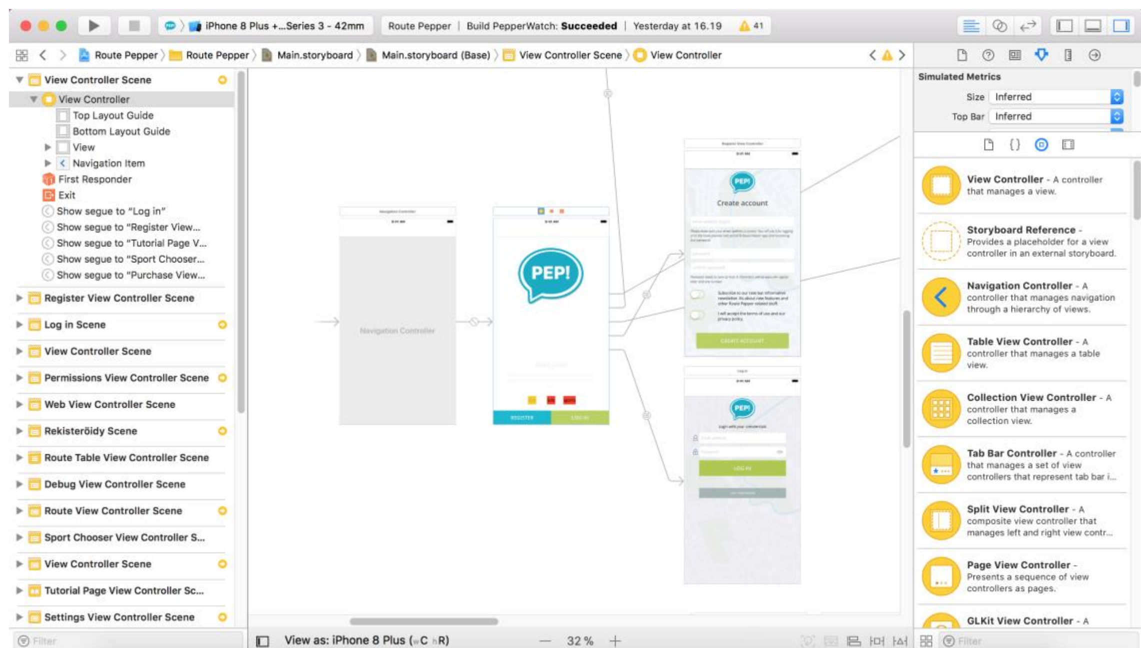

Figure 10. Storyboard window of the Route Pepper iOS application.

Simulators are a good way to test code without real devices. Xcode provides a wide variety of devices that can be simulated. Provided simulators are for iOS, watchOS and tvOS. Notifications can be simulated by providing a PushNotificationPayload.apns file, which includes the notification message, title and the notification category [7, p. 282-283]. Simulating locations and a walking courses is also possible, but is quite cumbersome to setup; a GPX file has to be created on a third-party website and given to Xcode to simulate [34].

Testing the application with real devices is always recommended, since it guarantees that the application actually works. When testing with real devices, permission has to be granted from the connected device. During the project, Xcode quite often had trouble recognizing the paired Apple Watch. Unplugging the iPhone and reattaching the cable usually solved the problem, but occasionally the devices or Xcode had to be booted.

Xcode offers a playground environment for the Swift programming language. The Swift playground is a way to experiment with code without creating an actual application. With just a few clicks, code can be tested without creating a project for it. Lines of code can

be written with the result immediately given without running any code. The result of each line is shown in the sidebar. Swift playground is useful for new developers learning the Swift language and for developers quickly want to experiment a piece of code or an algorithm outside of their application. [35.]

Xcode provides a built-in tool for GitHub integration via the Source Control tab. GitHub account can be linked to the project to give access to the remote repository. I personally liked to use git from the Mac console, but occasionally had difficulties with merging the Xcode project and its storyboard files. Some files got corrupted by missing lines. Using the Xcode's own source control might be a better option.

5.1.3 Google Firebase

Route Pepper runs as a serverless application utilizing Google Firebase. Google Firebase is a cloud service that offers real-time database, authentication, cloud functions and many other features [36]. The goal of Google Firebase is to allow the developers to focus on their products instead of worrying about deploying and managing servers. Google Firebase has a wide range of support, including web applications, iOS and Android [36]. For the Firebase to be used on an iOS application, the iOS project has to be added to the Firebase project through the Firebase Console, the necessary pod-files installed and the GoogleService-Info.plist file downloaded and added to the project. [37.] The Firebase documentation provides a good tutorial for doing so.

Real-time database

The Firebase Realtime Database is a cloud-hosted NoSQL JSON-document database used for storing and synchronization of data between the users in real-time [38.] The database is designed to synchronize data instantly between devices in real-time. The data is transferred via a WebSocket instead of traditional HTTP requests, allowing fast data synchronization. The synchronized data is also available offline via an on-device database. [39.] This means that the data is always available to the user even when the user loses connectivity. The offline mode is currently only available for Web, iOS and Android applications [38].

Authentication

Firebase Authentication provides secure user registration and authentication services. The service supports email and password accounts, Google, Twitter, Facebook and GitHub login with many more. [40.] Creating authentication services from scratch can be quite tedious. Using services like Firebase Authentication is a good way to provide secure authentication and lessen the project workload. Route Pepper uses email and password accounts on the web and the iOS application. Listing 5 displays how the Firebase Authentication service is used for Route Pepper. Authentication is not necessary for the Watch application since the user will be authenticated on the Route Pepper iOS application.

```
//User sign-in on iOS application
Auth.auth().signIn(with: credential) {(user,error) in
      if let error = error {
             //Handle error
      }
      //The user has signed in
      // ...
}

//User sign-out on iOS application
let firebaseAuth = Auth.auth()
do {
      try firebaseAuth.signOut()
} catch let sigOutError as NSError {
      //Handle error
}


//User registration on iOS application
Alamofire.request(http, method: .post, parameters: parametersFromUserInput,
encoding: JSONEncoding.default). response { response in

      if (response.response?.statusCode == 200 {
             //Successfully created the user
      } else {
             //Handle error
      }

}

//JavaScript code written for the Firebase Cloud Functions -service
admin.auth().createUser({
      email : emailFromJSON,
      password : passwordFromJSON
})
. then(function (userRecord) {
      //Successfully created the user
      //Handle the new user record by, for example, sending email or adding new
      data rows for the user

}
```

Listing 5. Using the Firebase Authentication on the iOS application.

Google Firebase has made authentication and user registration fairly simple after the necessary procedures have been made. The registration will be remembered even if the application is being used in offline-mode. For the user creation, the iOS application sends the user input to the Firebase Cloud Functions service where the data is handled, and the user record added to the user database. The request is made with Alamofire, which is a Swift-based HTTP-library.

Cloud Functions

Google Cloud Functions is a service that can host single-purpose JavaScript functions that are executed in a Node.js environment. The scripts written for the Cloud Functions will automatically run in response to Firebase features or HTTPS requests sent to the server. Automatic triggers will be emitted by Firebase features such as by the Realtime database, Firebase Authentication and Crashlytics. [41.] An application backend can simply be created using the Cloud Functions.

The benefits of using the Cloud Functions is that there is no need to setup and run servers. The Google application server automatically be scaled to fit the need and only the executed code will be billed [42]. The Cloud Functions are designed to be used alongside Firebase’s other services.

To use the Cloud Functions, Node.js has to be installed. After the installation, the Firebase Client has to be installed using npm and a sign-in with the Google account is required. After the installation and login, a local Firebase project has to be initialized. The code written for the Cloud Functions can be easily be deployed by running the ‘firebase deploy --only hosting’ command. [43.]

5.2 Communication model

The most important part of the project was the establishment of the connection between the Watch and iOS applications. The same data transfer and receive logic was used for each of the main features, excluding the Pep Point notifications. The data was received and sent by the AppDelegate on the iOS application and by the ExtensionDelegate on the Watch application.

The received data is first stored in public variables, after which the to-be-affected class was informed about the received data via the NotificationCenter class' observer structure created earlier in the application's lifecycle. The same logic was used for sending data to the other application but in reverse. When data was changed, the AppDelegate/ExtensionDelegate was informed by the change via an observer created during the launch of the application. The logic pattern is shown in Figure 11. The figure uses pseudocode to remain simple and generic.

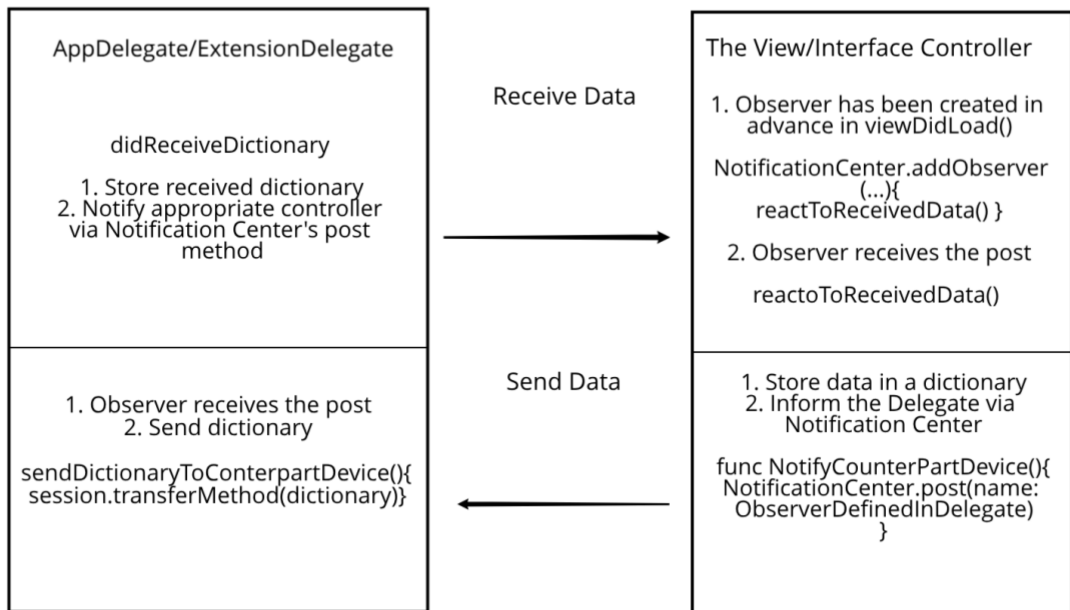

Figure 11. Communication logic for receiving and sending data between the applications.

5.3 The application

The application was made fairly simple both in its features and visual style. The application was meant to be more complex with a feature to start a route directly from the watch. This feature would have meant that the routes would have needed to be sent from the iOS application over to the Watch application, the routes displayed for the user and that the location services could have been started from the iOS application when a route was started. Starting the routes from the watch turned out to be too complex for our iOS application to handle. The feature was tested but the solution could not be implemented during the time allocated for the project, which lead us to postpone this

feature for later development. The feature is discussed more in the Future Development section of this chapter.

The first published version of the Watch application includes most of the main features of the iOS application: the changing of user settings, Pep Point creation and a unique way of displaying Pep Point notifications. The storyboard layout of the application is shown in the Figure 12.

Figure 12. Storyboard of the finished Route Pepper Watch application.

The main screen of the application is divided into two groups, nonRunningGroup and runningGroup (see Figure 12). The nonRunningGroup is shown when the Watch application has not received information of a started route from the iOS application. When a dictionary containing the route name and a Boolean value of true is received by the ExtensionDelegate, the nonRunningGroup will be hidden, whereas the runningGroup is made visible. The runningGroup displays the name of the started route, a label instructing the user to wait for Pep notifications and two action buttons: ‘Add Pep Point’ and ‘Settings’.

5.3.1 Pep Point notifications

The purpose of the Route Pepper application is to display the messages written for the Pep Point markers that the user has added to a route created with the Route Planner -tool. The Pep Point messages are voice dictated to the user upon entering a Pep Point location and a notification is also displayed.

If the user has an Apple Watch paired with the iPhone, all local iOS notifications will be displayed on the watch by default. When a notification scheduled by an iOS application is being displayed, it will automatically be forwarded to the Watch and processed by a class inheriting the WKUserNotificationInterfaceController class. The class can be used to customize the notification that will be shown to the user. The notifications shown on the iOS application only show the Pep Point message, whereas the notifications displayed on the watch additionally contain a map image, along with the Pep Point marker and the route line. Figure 13 displays the Pep Point notification received by the Watch application.

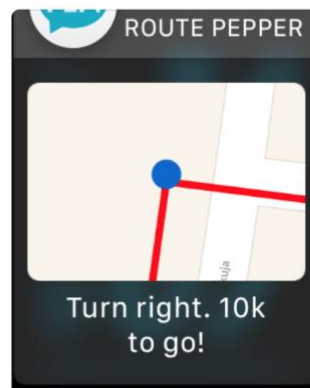

Figure 13. Pep Point notification on the Watch application.

Map Image

The image is generated by the iOS application when the user reaches the Pep Point location; just before scheduling the notification. The image creation is called using the createSnapshot(pepCoordinate:course:) method of MapSnapshotManager helper class (see Appendix 1). The method utilizes the MKMapSnapshotter class to take a snapshot of the area surrounding the Pep Point. The angle of the taken snapshot is determined by the user’s current location’s course. The course is the direction in which the device is traveling measured in degrees [44]. Options such as image size, span, scale and region

are also provided for the MKMapSnapshotter object. After the options are set, the snapshot is taken and forwarded to drawMarkers(snapshot:color:linewidth:pepCoordinate:) method for drawing the Pep Point marker and the route line on the taken snapshot image.

Drawing the marker and polyline

The drawing process on the snapshot image is declared by calling UIGraphicsBeginImageContent() method of the UIKit framework, which initiates the creation of graphical content. To draw on the image, the canvas area for the drawing has to be defined and the drawing started on the snapshot.

```
UIGraphicsBeginImageContext(snapshot.image.size) //start creation of graphics
let rectForImage = CGRect(x : 0, y : 0, width: snapshot.image.size.width,
height: snapshot.image.size.height) //canvas area
snapshot.image.draw(in :rectForImage) //start drawing on the image
```

The polyline is drawn based on the MKPolyline object received from the RouteViewController, the polyline-object is created from the trail's location. The trail-coordinates are iterated to align the coordinates with points on the snapshot image coordinates (see Appendix 1). After the exact points are determined, the route line width is set, and the determined points connected with a red line. The Pep Point marker is drawn with the drawAnnotation(snapshot:coordinates:context:) method (see Appendix 1). The reference to the context started earlier has to be provided so that the same graphics content will be drawn on. The method draws a blue ellipse on the location of the Pep Point. After both the Pep Point marker and the polyline has been drawn on the snapshot image, the new image will be fetched from the graphics content, the drawing process ended and the image returned back to the createSnapshot(pepCoordinate:course:) method for further processing.

```
let image = UIGraphicsGetImageFromCurrentImageContext()//the image is fetched
from the graphics context
UIGraphicsEndImageContext() //drawing process is ended
return image //the completed image is returned
```

The returned UIImage-object is turned into a JPEG Data-object with the UIImageJPEGRepresentation(_:_:) method. The result is returned back to the RouteViewController to be attached to the notification that will be scheduled.

Scheduling the notification

After the JPEG image has been created and returned by the MapSnapshotManager class, the returned image is attached to a local notification scheduled by the sendNotification(message:zone:imageData:) method. Listing 6 displays how the created image is attached to the notification payload. The created image Data object is put into the userInfo dictionary.

```
func sendNotification(message: String, zone: Int, imageData : Data? = nil) {

      ...

      let content = UNMutableNotificationContent() //Class for modifying the
local notification content

      ... //Add content such as title, message, sound, category

      if imageData != nil {
            content.userInfo = ["image" : imageData!]
      }

      ... //Schedule notification

}
```

Listing 6. Attaching the image to the notification payload.

The imageData variable is specified as optional since the same method is used for regular notifications (when the Watch application is not installed). Local notifications can be modified with the UNMutableNotificationContent class. The modified properties for the notification are the notification message, title, sound, notification category and the user info.

Displaying the notification on the Watch application

When the scheduled notification triggers while the user has a paired Apple Watch, the notification will automatically be displayed by the Watch. The notification will be received by the NotificationController, which will process the data in the notification payload (see Listing 7).

```
override func didReceive(notification: UNNotification, withCompletion
completionHandler : @escaping (WKUserNotificationInterfaceType) -> Void){
```

```
	let notificationBody = notification.request.content.body //message
	let dictionary = notification.request.content.userInfo
	if let imageData = dictionary[“image”] as? Data {
		let imageUIImage : UIImage = UIImage(data : imageData)! //turn
received image into an UIImage that can be displayed in the UI
		image.setImage(imageUIImage)//display Pep Point -image
	}

	messageLabel.setText(notificationBody)//Display Pep Point -message

	completion(.custom)

}
```

Listing 7. Handling the received notification to display the content to the user.

The received notification is stripped of off its contents, the Pep Point message set to the message label and the Data object is turned back into a UIImage that can be included in the long-look view of the notification. Finally, the notification is set to be shown as a custom notification. If the handling of the received notification takes too long to process, the notification will fall back to the static notification and the image will not be displayed in the notification.

5.3.2 Adding a Pep Point

Pep points are the main aspect of the Route Pepper application. Pep points are map coordinates that are created either on the Route Pepper Planner – web tool, on the iOS application or now on the Watch application. During application use, the points are added from the iOS application’s map screen but doing so while exercising was found to be too inconvenient, which is why the feature was added to the Watch application.

Location services

Adding Pep Points on the applications require location services to be available. The Pep Point coordinate is based on the user’s current location. Core Location is a framework used to obtain information on geographical location. To start the location services, the Watch application has to declare a CLLocationManager object and start the location services (see Listing 8). The interface controller has to conform to the CLLocationManagerDelegate. Classes conforming to the delegate have to implement the delegate methods.

```
class StartMenuInterfaceController: WKInterfaceController,
CLLocationManagerDelegate {

      ...

      let locationManager = CLLocationManager()

      override func awake(withContext context: Any?) {
            super.awake(withContext: context)
            locationManager.delegate = self //assigning the delegate
            locationManager.requestWhenInUseAuthorization() //request
            permission to use location services
            locationManager.desiredAccuracy = kCLLocationAccuracyBest //The
highest level of accuracy will be assigned

      ...

      }

      //Delegate methods that have to be implemented
      func locationManager(_ manager: CLLocationManager, didUpdateLocations
      locations: [CLLocation]) {
         //Handle updates
      }
      func locationManager(_ manager: CLLocationManager, didFailWithError
      error: Error) {
         //Handle errors
      }
```

Listing 8. Creation and activation of the CLLocationManager object.

The location manager object is created when the application is launched. When defining the location manager object, the authorization level and accuracy have to be provided. The user's location is only needed when the Pep Point is about to be created, which means that the requestWhenInUseAuthorization() is enough for the purpose. requestAlwaysAuthorization() can be used to regularly update the location while the application is running, but this would be unnecessary. Location accuracy is set to kCLLocationAccuracyBest which is the highest level of GPS accuracy.

Creating and sending the Pep Point

When the user presses the interface controller's 'Add Pep Point'-button, the method asks the initiated location manager for the user's current location. After the location has been determined, the method proceeds to a dialogue for adding the text for the Pep Point message using the pushTextInputController(withSuggestions:allowedInputMode:completion:) method (see Appendix 2).

The method presents a modal dialogue which allows the user to create the text input via voice dictation. The text could also be input via scribble (‘drawing’ the characters) or chosen from a set of suggested strings, but these are not used for the application. After the user is satisfied with the text, the text and the location data are stored inside a dictionary and sent to the iOS application via transferUserInfo(pepPoint:) for further processing (see Appendix 2). When the modal closes, an alert will be presented with presentAlert(withTitle:message:preferredStyle:actions:) method (see Appendix 2). The alert will notify the user whether the Pep Point creation succeeded or failed.

Receiving and pushing the Pep Point to Firebase

The Pep Point is received by the iOS application’s AppDelegate’s session(_:didReceiveUserInfo:) method. The Pep Point is stored in a public variable and the RouteViewController is notified of the received data. When the view controller’s observer receives the post, the controller’s createPepPoint(_message:_location:) method will be called, including the received Pep Point’s message and location data. Listing 9 displays how the Pep Point is pushed to Firebase and the annotation added to the iOS application’s map view.

```
func createPepPoint(_ message : String, _ location : CLLocationCoordinate2D){

        var pepfirref : DatabaseReference! //database reference
        pepfirref = self.ref.child(pepref).childByAutoId()//reference path to
a new child (id is auto generated)

        pepfirref.setValue(["message" : message, "language" : "fi_fi",
"speakOnEnter" : -2]) //set values for the new pep point
        pepfirref.child("location").setValue(["latitude": location.latitude,
"longitude" : location.longitude]) //set values for the new pep point
        let point = PepPointAnnotation(coordinate: location, name: message,
triggerradius: 50) //create annotation object
        self.mapView.addAnnotation(point)//add annotation to the mapView
    }
```

Listing 9. Pushing the Pep Point into Firebase and adding the annotation to the map view.

Calling self.ref.child(pepref).childByAutoId() creates the database reference to the new Pep Point and returns the reference (see Listing 9). After the creation, the message, coordinates and additional data is provided for the database reference. Finally, the Pep Point is added to the mapView object to be displayed on the user’s route map.

5.4 Sharing data between the applications

The application offers several settings for the user to customize their experience with the application. These settings include options for toggling voice guidance and notifications, option for the zone value (meters/feet) to determine at which distance the Pep Points are triggered, and options for the voice dictation: volume, speed and voice pitch. When the user has the Watch application installed, there will be a setting for changing the zoom level of the Pep Point map images sent to the watch via the notifications.

When a setting is changed on either one of the applications, the new value of the setting is stored in the device's user's defaults database. The database is a key-value database persistent across application launches. The new value is set using the UserDefaults class, with the help of a UserDefaultsManager struct [45].

After the value has been stored, a dictionary of the device's user defaults will be sent to the counterpart application via updateApplicationContext(dictionary) method. The dictionary contains all of the Route Pepper's application settings. This method is used since it only keeps the latest dictionary of values, overwriting the previous dictionary in the queue.

When the counterpart application receives the settings dictionary, the dictionary is forwarded to the SharedSettings class by calling it's setSettingsDictionary(dictionary:) method. The method iterates through the dictionary and adds the new application settings to the device's user's defaults database.

5.5 Testing

There was not much testing conducted for the Watch application, since it is heavily based on the features that have been tested for the iOS application. The iOS application has been tested by beta users, who have provided us with valuable feedback. The initial Watch application was created as an MVP, to see if there is demand for the application since we do not know how many of the iOS application's user will have an Apple Watch (both applications are released at the same time).

The Watch application was, however, tested in real-world use, with real devices, to see that it actually behaves as well as the iOS application. The ‘add Pep Point’ feature was tested to see if the Apple Watch’s GPS was accurate enough, and that the notifications arrive correctly (that it arrives on time and that the snapshot angle is rotated correctly).

The Pep Point notifications needed the most testing since the correct zoom level for the map snapshots was difficult to determine. The correct level had to be manually found through testing. The level had to be just right for the map image to be of any use. Through testing, it was found that a low zoom level was better in city environment, whereas a more zoomed out image was better in suburbs. The user needs more detailed information of the Pep Point surroundings in complex environments. Due to testing results, an option for adjusting the zoom level was added to the application.

5.6 The encountered problems

During the Watch application development several problems where encountered, some of which lead to compromises with the planned features. Most of these problems were due to the restrictions of the watchOS and the lack or restriction of frameworks. Problems encountered were with the understanding of the Watch Connectivity framework, sending and attaching map images to the notifications, starting the route directly from the watch and the creation of a map interface.

Application communication

Establishment of the communication between the applications was the most important part of the project. The Watch Connectivity framework is essentially a simple framework to learn and use, but there still were some obstacles that caused problems and confusion. Problems were encountered with both the background and the instant transfers.

While testing application communication with the background transfers, I somehow could not figure out why the data was occasionally received immediately, while sometimes it could take up to several minutes or till application launch. The cause for the issue was of course that the data sent in the background will be handled when the device decides that it has the time to process it.

When trying out a solution for the late handling of the background transfers, I tried out the sendMessage(_:replyHandler:errorHandler:) method for instant transfers. The method wakes up the counterpart application and the data will be handled immediately. This did not work out however, since apparently the method behaves differently when using it to send data from the iOS application to the Watch application. Sending an instant message to the Watch application will not wake up the application.

Starting a route on the Watch application

We wanted to make a feature to start a route directly from the Watch application. The feature was developed and tested, but the testing revealed serious problems. The problems lead us to drop the feature from the initial launch of the application, but it is scheduled for further development. This feature is discussed in the Further Development section.

The problems were due to the iOS application being built without the knowledge of a Watch application being added to the project in the future. The functions related to the starting of a route were too confined into the iOS application's view controllers to be accessed from the outside. The tested method was to a loop through all the necessary view controllers (to run the functions and start the location manager) when the message to start a route was received by the iOS application's AppDelegate. This did not work however, since occasionally the execution would be slow or even crash the application.

The reason for the problem was not found, but it might have been related to the tested method being too much for the application to handle; changing the application scenes while the application was not even launched in the first place. The routes were supposed to be able to be started even when the Route Pepper application was not running on the iOS.

The map interface

We wanted to have an interface that showed a live map with the route trail and the Pep Points for the user, which would have been good feature. When I was about to create the interface using the MapKit framework, I encountered a problem with the framework's restrictions on the watchOS. The WKInterfaceMap element was restricted to only have

five annotations at the same time and the trail drawing was also impossible. The map could only be used to show a specific region and to add a few annotations.

To be able to create a similar map view that the Route Pepper iOS application has, a custom map view would have to be created from scratch. The SpriteKit framework could be used to create such a view, but due to the complexness and the time restrictions, the idea was postponed. The feature was dropped and replaced with the sending of Pep Point map images with the notifications.

Pep Point notifications

The map interface feature was dropped and replaced with a feature of sending Pep Point map images to the Watch application via notifications. Creating this feature as a compromise did not work out without complications either.

To display the map snapshot images on the Watch application, I first tried to attach the images to the notification payload via the UNMutableNotificationContent class' attachment property. For the attachment to be shown in the notification, the image file also had to be stored on the Watch application, which meant that the file had to be on the device before the notification was received. This proved to be very problematic since the images could only be sent to the Watch application via the transferFile(_:metadata:) method, which sends it in the background and could take a while till it arrives. This also meant that if a Pep Point image was sent to the watch too early, the image could not have been rotated to match the user's current direction.

To solve this problem, I realized that an image could actually be compressed into a Data-object with the UIImageJPEGPresentation(_:_:) method. The object could then be stored into a dictionary as any other variable types. The dictionary could then be included with the notification in its userInfo content. This solution did not even cross my mind since every tutorial and solution I saw online was related to attaching an actual image file onto the notification.

5.7 Further development

During the application development, several obstacles where encountered, which hindered the creation of some of the feature goals that were planned for the finished application. Several features are scheduled for further development.

Starting a route from the watch

To fix the problems we had with starting a route from the Watch application, we would have to detach the Firebase functions and location management from the view controllers and move them to their own separate helper classes. This would mean that the routes could be fetched from the database and the location services started from the iOS application's AppDelegate without accessing these functions from a view controller. To start a route from the watch, the available routes would have to be listed for the user to choose from. The routes would have to be sent from the iOS application.

Standalone Watch application

Apple Watch Series 3 introduced a model with a sim-card slot for establishing its own cellular network. For Route Pepper, this would mean that the watch could make its own Firebase connection to fetch routes and add Pep Points. For Route Pepper to be used without the iPhone close by, the application has to be modified so that the watch can connect to the Firebase by itself. The watchOS currently does not have Firebase support, but the connection could still be established with the NSURLSession class by connecting to the Firebase Database REST API [46, 47].

6 Conclusion

The Bachelor's Thesis covered the basics of Apple Watch development by exploring its features, application hierarchy and frameworks. The goal of the featured project was to create a Watch application as an extension to the Route Pepper iOS application. The application goals were to implement the iOS application's main features to be accessed from the Watch application: change of application settings, starting a route, adding Pep Points and a map interface. Some of these features could not be completed during time

allocated for the project and on some features the team had to compromise to get around the Apple Watch's restrictions.

The features were created using the Core Location framework for the location services, Watch Connectivity framework for communication between the applications, and the WatchKit framework to create the actual Watch application. The establishment of the communication became the key part of the project, since the two-way communication was required for most of the features.

During the project, the author encountered several problems that hindered the creation of some of the features. Implementing a similar map interface from the iOS application was deemed impossible to be create during the project timespan; the map element provided by the WatchKit framework proved to be too limited for our needs. A similar view could possibly have been created with the SpriteKit framework, but that was postponed for later development. The feature was however replaced by sending map images of the Pep Point locations to the application via notifications.

The feature of starting the route directly from the watch also proved to be too troublesome. The iOS application was developed in a way, that made it difficult to integrate the Watch application into it. Some of the functionality was too confined into the application's views. The feature was left out from the initial release of the Watch application but will definitely be developed in the future.

All of the parties involved in the project were satisfied with the project results. The Watch application reached most of its feature goals and it was published in the Apple App Store to be used as an extension to the iOS application. Both the author and the project manager learned from the conducted project, and the Watch application will be further developed in the future.

References

1 Lamkin, Paul. 2017. Wearable Tech Market To Double By 2021. Web Document. Forbes. <https://www.forbes.com/sites/paullamkin/2017/06/22/wearable-tech-market-to-double-by-2021/#1cd6a01ad8f3>. Accessed 18 April 2018.

2 Page, Carly. 2015. Apple Watch is 'world's best selling wearable' with 4.2 million shifted is Q2. Web Document. The Inquirer. <https://www.theinquirer.net/inquirer/news/2418517/apple-watch-is-worlds-best-selling-wearable-with-42-million-shifted-in-q2>. Accessed 2 April 2018.

3 Silbert, Sarah. 2018. Apple Watch: Everything You Need to Know. Web Document. Lifewire. <https://www.lifewire.com/apple-watch-models-4151590>. Accessed 2 April 2018.

4 Fingas, Roger. 2016. Apple Watch runs 'most' of iOS 8.2, may use A5-equivalent processor. Web Document. AppleInsider. <http://appleinsider.com/articles/15/04/23/apple-watch-runs-most-of-ios-82-may-use-a5-equivalent-processor>. Accessed April 2 2018.

5 Horwitz, Jemery. 2015. Apple announces watchOS 2 with third-party Apple Watch apps, new Timepieces, video playback, much more. Web Document. 9to5Mac. <https://9to5mac.com/2015/06/08/apple-announces-new-version-of-watchos/>. Accessed April 2 2018.

6 Haslam, Karen. 2018. watchOS 4 latest update & new Apple Watch features. Web Document. Macworld. <https://www.macworld.co.uk/news/apple/watchos-4-latest-update-new-apple-watch-features-3640975/>. Accessed April 4 2018.

7 Amer, Ehab; Atkinson, Scott; Azarpour, Soheil; Morey, Matthew; Morrow, Ben; Tam, Audrey & Wu, Jack. 2017. watchOS by Tutorials: Making Apple Watch apps with watchOS 4 & Swift 4. E-book. raywenderlich.com Tutorial Team.

8 App Programming Guide for watchOS: Context Menus. 2015. Web Document. Apple Developer. <https://developer.apple.com/library/content/documentation/General/Conceptual/WatchKitProgrammingGuide/Menus.html>. Updated 12 December 2016. Accessed 6 April 2018.

9 Local and Remote Notifications Programming Guide: Local and Remote Notifications Overview. 2009. Web Document. Apple Developer. <https://developer.apple.com/library/content/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG>. Updated 13 November 2017. Accessed 8 April 2018.

10 Local and Remote Notifications Programming Guide: APNs Overview. 2009. Web Document. Apple Developer. <https://developer.apple.com/library/content/documentation/NetworkingInternet/C

onceptual/RemoteNotificationsPG/APNSOverview.html>. Updated 13 November 2017. Accessed 8 April 2018.

11 App Programming Guide for watchOS: Notification Essentials. 2015. Web Document. Apple Developer. <https://developer.apple.com/library/content/documentation/General/Conceptual/WatchKitProgrammingGuide/BasicSupport.html>. Updated 12 December 2016. Accessed 10 April 2018.

12 App Programming Guide for watchOS: Complication Essentials. 2015. Web Document. Apple Developer. <https://developer.apple.com/library/content/documentation/General/Conceptual/WatchKitProgrammingGuide/ComplicationEssentials.html>. Updated 12 December 2016. Accessed 26 April 2018.

13 App Programming Guide for watchOS: The Watch App Architecture. 2015. Web Document. Apple Developer. <https://developer.apple.com/library/content/documentation/General/Conceptual/WatchKitProgrammingGuide/DesigningaWatchKitApp.html>. Updated 12 December 2016. Accessed 13 April 2018.

14 Baumgartner, Justin. WatchKit 101: Everything you need to know about the anatomy of Apple WatchKit. Web Document. Solstice. <https://www.solstice.com/blog/watchkit-101-everything-you-need-to-know-about-the-anatomy-of-apple-watchkit>. Accessed 7 May 2018.

15 WatchKit. Web Document. Apple Developer. <https://developer.apple.com/documentation/watchkit>. Accessed 7 May 2018.

16 WKExtensionDelegate. Web Document. Apple Developer. <https://developer.apple.com/documentation/watchkit/wkextensiondelegate>. Accessed 8 May 2018.

17 applicationDidFinishLaunching(). Web Document. Apple Developer. <https://developer.apple.com/documentation/watchkit/wkextensiondelegate/1628241-applicationdidfinishlaunching>. Accessed 8 May 2018.

18 applicationDidBecomeActive(). Web Document. Apple Developer. <https://developer.apple.com/documentation/watchkit/wkextensiondelegate/1628185-applicationdidbecomeactive>. Accessed 8 May 2018.

19 applicationDidEnterBackground(). Web Document. Apple Developer. <https://developer.apple.com/documentation/watchkit/wkextensiondelegate/1650865-applicationdidenterbackground>. Accessed 8 May 2018.

20 applicationWillEnterForeground(). Web Document. Apple Developer. <https://developer.apple.com/documentation/watchkit/wkextensiondelegate/1650868-applicationwillenterforeground>. Accessed 8 May 2018.

21 WKExtensionDelegate. Web Document. Apple Developer. <https://developer.apple.com/documentation/watchkit/wkextensiondelegate>. Accessed 9 May 2018.

22 Watch Connectivity. Web Document. Apple Developer. <https://developer.apple.com/documentation/watchconnectivity>. Accessed 15 April 2018.

23 WCSession. Web Document. Apple Developer. <https://developer.apple.com/documentation/watchconnectivity/wcsession>. Accessed 15 April 2018.

24 transferUserInfo(_:). Web Document. Apple Developer. <https://developer.apple.com/documentation/watchconnectivity/wcsession/1615671-transferuserinfo>. Accessed 16 April 2018.

25 updateApplicationContext(_:). Web Document. Apple Developer. <https://developer.apple.com/documentation/watchconnectivity/wcsession/1615621-updateapplicationcontext>. Accessed 16 April 2018.

26 transferFile(_:metadata:). Web Document. Apple Developer. <https://developer.apple.com/documentation/watchconnectivity/wcsession/1615667-transferfile>. Accessed 16 April 2018.

27 sendMessage(_:replyhandler:errorHandler:). Web Document. Apple Developer. <https://developer.apple.com/documentation/watchconnectivity/wcsession/1615687-sendmessage>. Accessed 16 April 2018.

28 WKExtension. Web Document. Apple Developer. <https://developer.apple.com/documentation/watchkit/wkextension>. Accessed 18 April 2018.

29 Singh, Amit. 2003. What is Mac OS X? Web Document. osxbook.com. <http://osxbook.com/book/bonus/ancient/whatismacosx/history.html>. Accessed 22 April 2018.

30 Swift Has Reached 1.0. 2014. Web Document. Apple Developer. <https://developer.apple.com/swift/blog/?id=14>. Accessed 22 April 2018.

31 Nyisztor, Károly. 2017. Why Learn Swift? The creation and evolution of a new programming language. Web Document. Pluralsight. <https://www.pluralsight.com/blog/software-development/swift-history>. Accessed 10 April 2018.

32 Solt, Paul. 2015. Swift vs. Objective-C: 10 reasons the future favors Swift. Web Document. InfoWorld. <https://www.infoworld.com/article/2920333/mobile-development/swift-vs-objective-c-10-reasons-the-future-favors-swift.html>. Accessed 22 April 2018.

33 Sakaimbo, Nicholas. 2017. Storyboards Tutorial for iOS: Part 1. Web Document. raywenderlich.com. <https://www.raywenderlich.com/160521/storyboards-tutorial-ios-11-part-1>. Accessed 23 April 2018.

34 Augusteo, Victor. 2017. How To Simulate GPS Movement in iOS. Web Document. Medium. <https://medium.com/@augusteo/how-to-simulate-gps-movement-in-ios-cca61907df2c>. Accessed 30 April 2018.

35 Elliott, Keith. 2016. Swift Playgrounds–Interactive Awesomeness. Web Document. Medium. <https://medium.com/swift-programming/swift-playgrounds-interactive-awesomeness-2a74143c233>. Accessed 23 April 2018.

36 Overview. Web Document. Firebase. <https://firebase.google.com/docs/>. Accessed 7 May 2018.

37 Add Firebase to your iOS Project. Web Document. Firebase. <https://firebase.google.com/docs/ios/setup>. Accessed 7 May 2018.

38 Dufetel, Alex. 2017. Introducing Cloud Firestore: Our New Document Database for Apps. Web Document. Firebase. <https://firebase.googleblog.com/2017/10/introducing-cloud-firestore.html>. Accessed 7 May 2018.

39 Esplin, Chris. 2016. What is Firebase? Web Document. How To Firebase. <https://howtofirebase.com/what-is-firebase-fcb8614ba442>. Accessed 7 May 2018.

40 Firebase Authentication. Web Document. Firebase. <https://firebase.google.com/docs/auth/>. Accessed 8 May 2018.

41 Cloud Functions for Firebase. Web Document. Firebase. <https://firebase.google.com/docs/functions/>. Accessed 8 May 2018.

42 Eschweiler, Sebastian. 2017. Introduction to Firebase Cloud Functions. Medium. <https://medium.com/codingthesmartway-com-blog/introduction-to-firebase-cloud-functions-c220613f0ef>. Accessed 8 May 2018.

43 Get Started: Write and Deploy Your First Functions. Web Document. Firebase. <https://firebase.google.com/docs/functions/get-started>. Accessed 8 May 2018.

44 CLLocation. Web Document. Apple Developer. <https://developer.apple.com/documentation/corelocation/cllocation>. Accessed 12 May 2018.

45 UserDefaults. Web Document. Apple Developer. <https://developer.apple.com/documentation/foundation/userdefaults>. Accessed 12 May 2018.

46 NSURLSession. Web Document. Apple Developer. <https://developer.apple.com/documentation/foundation/nsurlsession>. Accessed 13 May April 2018.

47 Firebase Database REST API. Web Document. Apple Developer. <https://firebase.google.com/docs/reference/rest/database/>. Accessed 13 May 2018.

MapSnapshotManager class

```
import Foundation
import UIKit
import MapKit
import UserNotifications
import UserNotificationsUI

class MapSnapshotManager {

    var mapView : MKMapView? = nil
    var polyline : MKPolyline? = nil

    let si = SharedSettings.sharedInstance

    init(_ mapView : MKMapView, _ polyline : MKPolyline){

        self.mapView = mapView
        self.polyline = polyline
    }

    func createSnapshot(pepCoordinates : CLLocationCoordinate2D, course :
Double, result: @escaping (_ data : Data?) -> Void) {

        let options = MKMapSnapshotOptions()

        let span = Double(si.ud.imageZoomOption)
        options.region = MKCoordinateRegion(center: pepCoordinates, span:
MKCoordinateSpan(latitudeDelta: span, longitudeDelta: span))
        options.size = (self.mapView?.frame.size)!
        options.scale = 2.0
        options.camera.heading = course

        let snapshotter = MKMapSnapshotter(options: options)
                snapshotter.start { snapshot, error in
                    guard let snapshot = snapshot else {
                        result(nil)
                        return
                    }
                    let image = self.drawMarkers(snapshot, UIColor.red, 10,
pepCoordinates)
                    if let imageData = UIImageJPEGRepresentation(image, 1.0){
                        result(imageData)
                    }
        }
    }


    func drawMarkers(_ snapshot : MKMapSnapshot, _ color: UIColor, _
lineWidth: CGFloat, _ coordinates : CLLocationCoordinate2D) -> UIImage {

        UIGraphicsBeginImageContext(snapshot.image.size)

        let rectForImage = CGRect(x: 0, y: 0, width:
snapshot.image.size.width, height: snapshot.image.size.height)

        snapshot.image.draw(in: rectForImage)

        var pointsToDraw = [CGPoint]()

        let points = self.polyline!.points()
        var i = 0
        while (i < self.polyline!.pointCount)  {
```

```
            let point = points[i]
            let pointCoord = MKCoordinateForMapPoint(point)
            let pointInSnapshot = snapshot.point(for: pointCoord)
            pointsToDraw.append(pointInSnapshot)
            i += 1
        }

        let context = UIGraphicsGetCurrentContext()
        context!.setLineWidth(lineWidth)


        for point in pointsToDraw {
            if (point == pointsToDraw.first) {
                context!.move(to: point)
            } else {
                context!.addLine(to: point)
            }
        }

        context?.setStrokeColor(color.cgColor)
        context?.strokePath()

        drawAnnotation(snapshot, coordinates, context: context!)

        let image = UIGraphicsGetImageFromCurrentImageContext()
        UIGraphicsEndImageContext()
        return image!
    }


    func drawAnnotation(_ snapshot : MKMapSnapshot, _ coordinates :
CLLocationCoordinate2D, context : CGContext!){
        let shapeSize = 40.0
        let point: CGPoint = snapshot.point(for: coordinates)
        let pointX = Double(point.x)
        let pointY = Double(point.y)
        let x = pointX - (shapeSize * 0.5)
        let y = pointY - (shapeSize - 10)

        let rect = CGRect(x: x, y: y, width: shapeSize, height: shapeSize)

        let blue = UIColor(red: 0.0/255.0, green: 100.0/255.0, blue:
207.0/255.0, alpha: 1.0)


        context.addEllipse(in: rect)
        context.setFillColor(blue.cgColor)
        context.fillPath()

    }
```

Add Pep Point method on Watch Application

```
@IBAction func addPepPoint(){
        self.locationManager.requestLocation()
        var failure = false
        var errorMessage = "Couldn't add Pep Point"
        if let location = self.locationManager.location {
            var message : String = ""
            presentTextInputController(withSuggestions: nil, allowedInputMode:
      WKTextInputMode.plain, completion: {(results) -> Void in
            if results != nil && results!.count > 0 && (running["running"] as!
            Bool == true){
                        message = results?[0] as! String

                        let pepPoint : [String : Any] = ["pepPoint" :
["message" : message, "location" : ["latitude" : location.coordinate.latitude
as Double, "longitude" : location.coordinate.longitude as Double]]]

                     let session = WCSession.default
                     session.transferUserInfo(pepPoint)

                } else {
                    errorMessage = "Couldn't add Pep Point"
                    failure = true
                }

                if (failure == true) {

            let action = WKAlertAction(title: "Dismiss", style: .cancel) {}
                    self.presentAlert(withTitle: "Error", message:
errorMessage, preferredStyle: .alert, actions: [action])

                } else {

                    let action = WKAlertAction(title: "Dismiss", style:
.cancel) {}
                    self.presentAlert(withTitle: "Success", message: "Pep
Point was added", preferredStyle: .alert, actions: [action])
                }
            })
        } else {
            let action = WKAlertAction(title: "Dismiss", style: .cancel) {}
            self.presentAlert(withTitle: "Error", message: "Couldn't find user
location", preferredStyle: .alert, actions: [action])


        }
```