Expertise
and insight
for the future

Glafira Privalova

Technology behind watchOS applications for Apple Watch

Metropolia University of Applied Sciences

Bachelor of Engineering

Software Engineering

Bachelor's Thesis

30 May 2019

Author Title Number of Pages Date	Privalova Glafira Technology behind watchOS applications for Apple Watch 29 pages 30 May 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Kari Salo, Pricipal Lecturer

The purpose of this study was to explore the evolution, capabilities and development challenges of smartwatch applications. This thesis focuses on application development for Apple Watch and what the platform offers for efficient user experience.

Although Apple Watch has plenty of competitors in the product category, it greatly benefits from its seamless interaction with other Apple devices. Its main goal is providing glanceable and readily available information to users. The device receives local and remote notifications received by paired iOS devices out-of-the-box. The hardware is equipped with several sensors that are used to collect health and fitness related data making Apple watch one of the most personal devices.

The methods used for gathering information for this project were a thorough study of Apple's official documentations along with development of experimental projects. Exploring the platform and its development is significant as Apple Watch highly influences the field. Apple wearables hold approximately half of the market in the sector.

As a result of this thesis, the author concludes that Apple Watch platform offers a great complement to Apple's interconnected ecosystem. The platform's easy integration with iOS and MacOS applications produces convenient and complete user experience.

Keywords	iOS, watchOS, software development, smartwatch, Apple Watch, WatchKit, Swift

Contents

Figures and Tables

List of Abbreviations

iOS	Apple iPhone Operating System
watchOS	Apple Watch Operating System
NFC	Near-field communication
WWDC	Apple Worldwide Developers Conference
eSIM	Embedded sim card
LED	Light-emitting diode
App	Application
GUI	Graphical User Interface
NFC	Near-field communication
LCD	Liquid crystal display
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure, an extension of the Hypertext Transfer Protocol.

1 Introduction

The aim of this study is to explore smartwatches focusing on Apple Watch and the platform's capabilities. From the first personal computer in 1975 and the first mobile cell phone not long after, portable digital devices have become key part of the modern life [1, 2]. Today, we have the smallest wearable devices that are capable of recording videos, receiving and sending messages, making payments through NFC (Near-field communication) technology and tracking significant health and fitness related information.

Apple Inc. is leading the market for smartwatches holding approximately 50 percent of sales for the past few years [3]. The company has deeply integrated product lines commonly called the Apple ecosystem. Apple Watch, along with iPhone, iPad, Mac and several other product categories of the company, provides seamless overall experience to the end user.

Apple Watch applications are a valuable extension to iOS (Apple iPhone Operating System) applications. They offer glanceable and concise information at hand. Although standalone Watch applications cannot exist at the moment, these extensions are readily available with their counterpart iOS application in the App Store.

The development process for Watch and iOS applications has similar traits. Developers familiar with the iOS platform have found Watch application development closely related and intuitive.

This thesis study explores the process and challenges in Watch application development. It covers Watch applications' main features such as notifications, user interface and data sharing with the counterpart iOS application. In addition, the evolution of watchOS (Apple Watch Operating System) along with all the changes it brings to the development are thoroughly discussed.

2 History behind the technology

2.1 Smartwatches

The world's first smartwatch was a digital electronic watch named Pulsar. This Pulsar Time Computer was introduced in 1972 by the Swiss watch manufacturer Hamilton Watch Company. [4].

Hamilton's inspiration came from a small project completed two years prior for a futuristic Hollywood film "2001: A Space Odyssey". Following the request of director Stanley Kubrick and science fiction writer Arthur Clarke, Hamilton created two "Odyssey" clocks and several watches used in the film production. [4].

Pulsar was equipped with LED (light-emitting diode) display or "time screen" to use Hamilton's term which showed three or four digits to represent the hour and minutes upon pressing a button on right side of the watch. A long-press of the button would switch the digits from hours and minutes to show only seconds. The display automatically turns off after displaying the time roughly for one minute. [4].

The user experience was unique and less intuitive compared to its unfair competition, analog quartz watches. The pulsar digital watch shows time on-demand (on press of a button) unlike its competitors with mechanical components pointing at the right time at all times. The battery depleting LED technology was later replaced by LCD (liquid crystal display) as the standard for digital watches. Although this digital watch was a success, selling about 400 pieces at $2100 price, competition later arose between 1972 and 1976 ultimately dropping digital watches to a price of $3 and lower. Consequently, there was a lack of interest from innovative companies to experiment and push the boundaries of wearable technologies. [4].

Some early smartwatches continued to revolutionize wearable technologies. The 1982 Pulsar NL C01 with internal memory capable of storing up to 24 digits and the 1984 Seiko RC-1000 carrying 2KB RAM capable of establishing wired connection with other hardware such as PCs, Apple computers and Commodore C64 were big milestones for the product category.

Although highly innovative, these devices lacked connectivity with the rest of the world. Preceding the invention of Bluetooth, the only option for such connection was using

cellular hardware which would be unrealistic to carry on wrist due to its enormous size at the time. The 1983 Motorola DynaTAC 800X was the world's first portable phone with cellular technology (see Figure 1). [5].

Figure 1. 1983 Motorola DynaTAC 800X cellular telephone

The first wirelessly connected smartwatch, Seiko Receptor, was introduced in 1990. It enabled paging to the watch by calling an automated service. The watch also came with a map containing regions where reception is supported as the FM sub-carrier signal used was available in limited geographic areas. [5].

Companies like Seiko, Samsung and IBM produced several short-lived smartwatches. Between 2004 and 2008, Microsoft had SPOT (Smart Personal Object Technology) smartwatches that used FM broadcasts for updates with yearly subscription that continued to send out updates until 2012. Most of these devices used FM wireless technology. In 2009 Samsung introduced Samsung S9110 Watch Phone followed by

Sony Ericsson's LiveView (2010) and Allerta's InPulse (2010), devices capable of connecting to a smartphone to give quick access to user's data right on the wrist. [5].

Apple introduced its mini-screen music player, iPod Nano, in 2011. Its technology may have inspired smartwatches such as VIMM One (2011) and Motoactv (2011). These devices were capable of music playback on the wrist.

Companies that are not necessarily digital technology centric, played an important role in the wearable digital devices' evolution. The appeal of wearables as fitness kit appealed Nike+ among others to manufacture a variety of smartwatches and health trackers. [5].

Samsung produced several Samsung Galaxy Gears (Gear 2, Gear 2 Neo and Gear Fit to name few) that run Tizen and Android OS (Operating System) some of which capable of running independently from a smartphone. Major innovations and developments focused on smartwatches hardware and software as a single product until Google's Android Wear in 2014.

The Android Wear which later was rebranded as Watch OS offered manufacturers to use this modified Android operating system in their smartwatches. Having this common operating system minimized the effort to build seamless connectivity with Android smartphones. [5].

Android Wear enabled developers to use the same libraries to develop for Android wear as they do for smartphones running Android. In addition to the ease of development, the operating system allowed designing for both circular and rectangle interfaces which gave flexibility to watch manufacturers and designers. [5].

2.2 Apple Watch

Apple introduced the Apple Watch, later predictably rebranded as the Apple Watch Series 1, along with watchOS in 2014. Despite the late launch into the market, Apple's first smartwatch had one of the biggest impacts for the product category as can be seen from the statistics of its market shares. The device paired with iPhone and other accessories such as headphones via Bluetooth and supported Wi-Fi internet connection. Health and fitness tracking were the main focus. The watch included Maps despite lack of onboard GPS rather fetching information from paired GPS-connected iPhone. [6].

The Apple Watch Series 2 was introduced in 2016 adding built-in GPS with waterproof feature. The dependency on paired iPhone remained for phone calls, Siri and other features. The device offered smoother interaction due to its upgraded processor (dual-core S2 chip) and new GPU for 60 fps (frame per second) screen refresh rate. [7].

Apple Watch Series 3, announced in 2017, broke the reliance on paired iPhone with its cellular LTE connectivity. Apps stayed connected regardless of Wi-Fi or nearby paired iPhone with incredibly fast processor, the dual-core S3 chip. This processor measured 70 percent faster than its predecessor according to Apple and conducted benchmarks. The watch also came with a new wireless chip called W2 that combines Wi-Fi and Bluetooth for faster performance. The series 3 was arguably the biggest upgrade for apple watches and the device category. [8].

Apple Watch Series 4 (2018) improved the existing features adding electric heart sensor along with ECG (electrocardiogram) app that can collect medical data further supporting the health and fitness aspect of the device. [9].

3 User Experience

All iOS applications that receive local or remote notifications are guaranteed to appear on user's wrist. WatchOS receives notifications from a paired iOS device regardless of an associated Watch App [10]. This feature made Apple Watch a useful device even before existence of Watch applications. In addition, the device contains convenient built-in applications that provide good user experience.

3.1 Notifications

WatchOS determines whether or not to show notifications on the watch based on several criteria that are not under developers' control. As a simple rule, notifications will appear on the watch if the iOS device is locked and the watch is on user's wrist. [11].

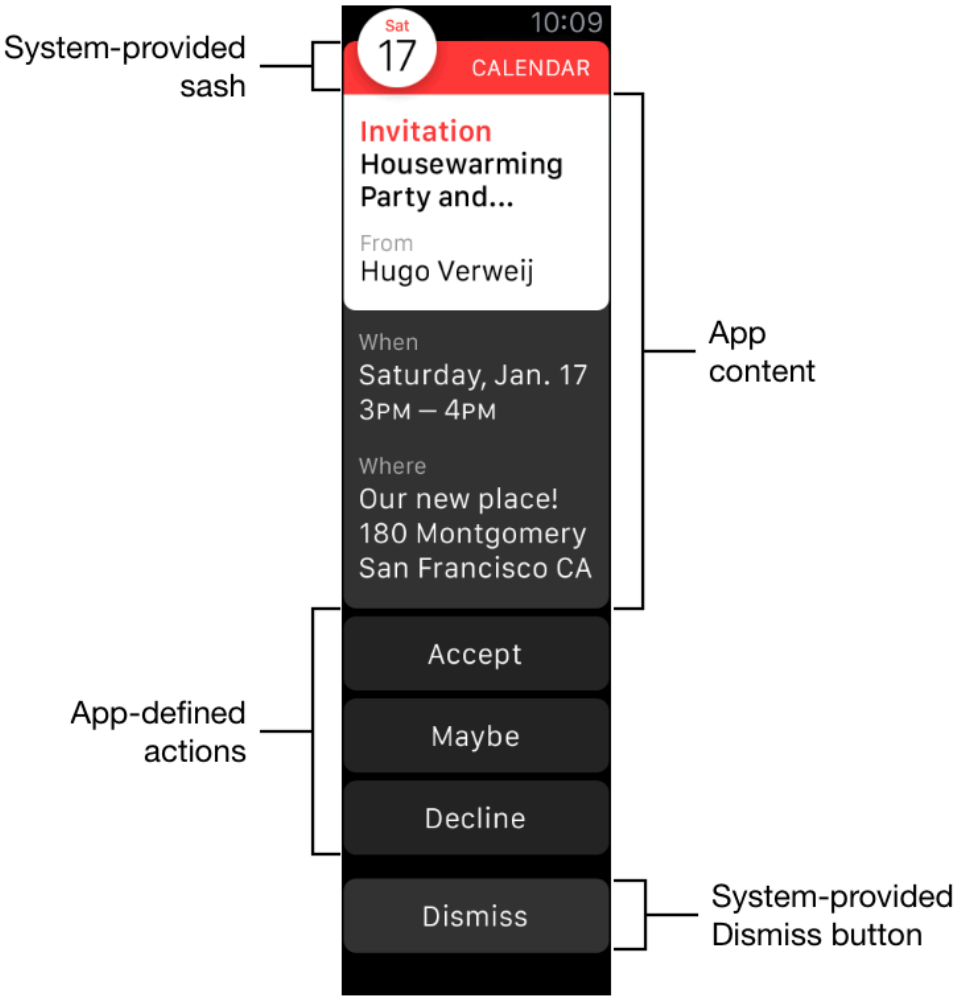

Figure 2. A long-look notification interface [11].

The above notification displaying behavior holds true for both remote notifications received from external source and local notifications triggered from the iOS or watchOS application. Local notifications triggered by either the iOS or watchOS application are handle in identical manner. They are sent to the iOS application and internally determined whether to display on the watch or iOS device screen. [11].

Once the notification is received by the watch, WatchKit is in charge of displaying it to the user. It can present the default handler of the notification and might give the accompanying Watch App a chance to customize the look and actions of the notification. [11].

3.2 Complications

Apple Watch offers a selection of watch faces. The watch shows a user selected watch face as the default display shown upon glances. Watch faces contain additional glanceable information such as date, temperature, images inside interface boxes called *complications*.

Complications are the small informative interfaces users can add to their preferred watch face. “The term *complication* comes from watchmaking, where the addition of features added complexity to the watch construction”, states the official documentation of Apple Watch. [12]. This interface components have three useful features when in use:

- Present glanceable latest information of the given application (e.g. the temperature for a weather app *complication*).
- Provide quick entrance point to the application. Tapping on a *complication* will open the associated application from its suspended state.
- Allow the application to execute longer background operations.

Complications become visible when the user glances at the watch. Therefore, applications that support *complications* must provide the latest data in the background, before they become visible. Tapping on a *complication* launches the application immediately. This is possible because watchOS keeps applications that have been placed on the active watch face suspended in memory, ready to launch at any point. Although, implementing *complications* is not a requirement, supporting them provides valuable advantages and is recommended by Apple. [12].

Complications differ by sizes and shapes, forming “families”. Currently, there are ten different *complication* families introduced by Apple (see Figure 5). However, this number will undoubtedly change with newer watch models. For example, the last Apple conference introduced Watch Series 4 with extra gauge corner *complications* (see Figure 3). Starting from Watch Series 2, watches support at least one *complication* on their watch faces (refer to examples in Figure 3 and Figure 4). [13]. It is always beneficial to support as many *complication* families as possible.

Figure 3. Watch faces with *complications* on Apple Watch series 0 to 4

Figure 4. Complication families introduced in watchOS 2

In order to take advantage of complication, the first requirement is to ensure the system recognizes the app's support for *complications*. This can be done by adding "*Complications Configuration*" section in target settings of the WatchKit Extension bundle. WatchOS needs to recognize the Watch App as *complication* provider which is achieved by registering the *complication* data source in configurations. The settings contain the list of all currently available *complication* families. Applications select *complication* families by simply placing checkmarks in Xcode configuration GUI (Graphical User Interface). The configuration with support to all currently available *complication* families can be seen in Figure 5.

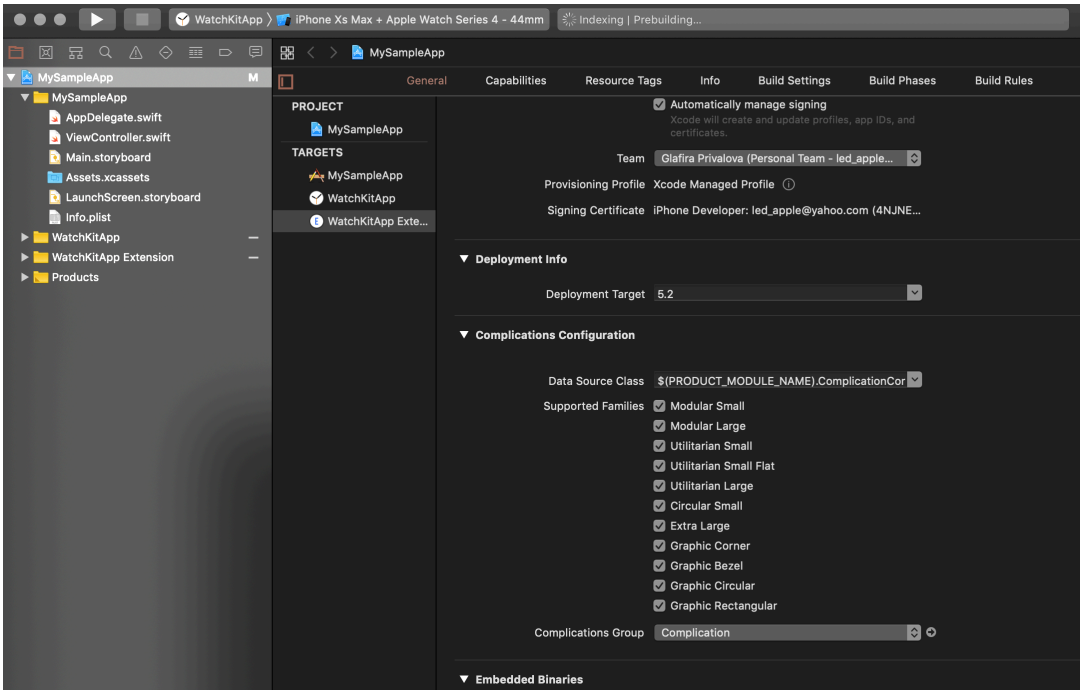

Figure 5. Complications configuration in WatchKit extension settings

Additionally, Xcode supports automatic generation of *complication* configurations. The "*include complication*" flag is defining this feature. It can be enabled when adding WatchKit App into an existing iOS project as shown in Figure 6. [15].

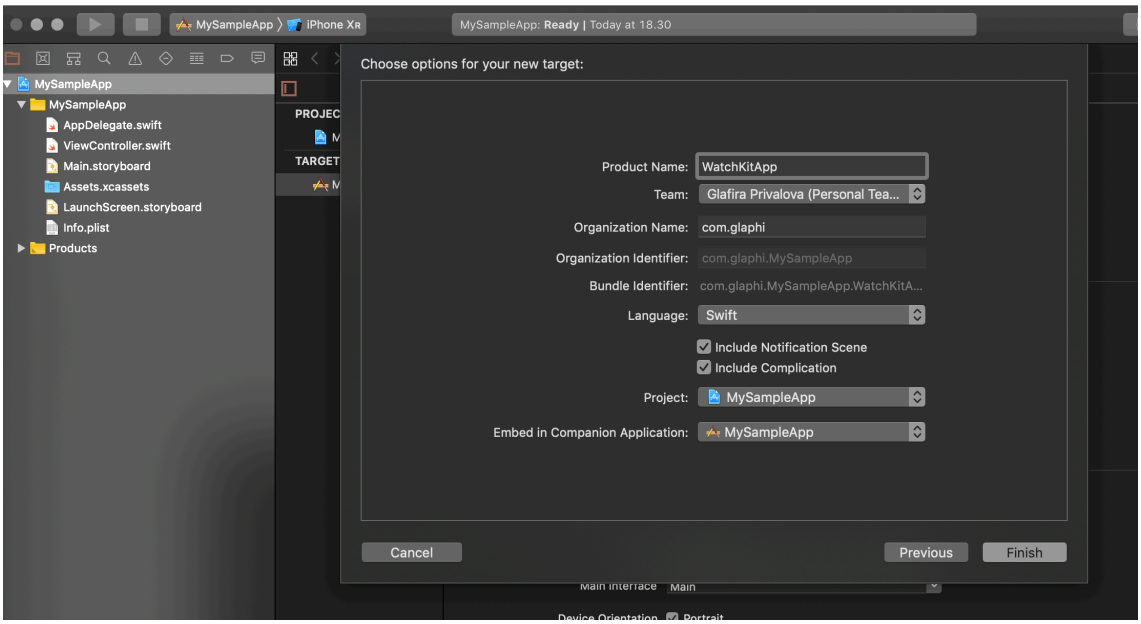

Figure 6. Configuration dialogue when adding WatchKit App target into iOS project

The information needed to display the *complication* is cached and rendered in advance. WatchOS expects all applications with *complication* support to provide the necessary

data source. A data source protocol, `CLKComplicationDataSource`, is defined in the ClockKit framework for this purpose. This data source readily offers the application specific data along with time and date information for each watch face template the platform provides. [15].

A calendar application may display the name of the next event inside a *complication* interface. The system will determine if this event is timely and worth displaying by using the linked time and date information.

Providing timely and useful information is the main goal of *complications*. Applications show the current, previous or future information depending on their internal logic. The previous example application, a calendar app, may choose to show the last and the upcoming events information as shown in Listing 1.

```
// MARK: - CLKComplicationDataSource

// Called by the system to request time travel direction.
func getSupportedTimeTravelDirections(
   for complication: CLKComplication,
   withHandler handler:
   @escaping (CLKComplicationTimeTravelDirections) -> Void) {

    // Support future and previous information.
    handler([.forward, .backward])
}
```

Listing 1. Complication information direction.

Complications allow securing the displayed information. There are times where the information in an application's *complication* may be private. The framework checks the security type of the information before displaying on the watch face. This is demonstrated in Listing 2.

```
// Called by the system to determine privacy behavior.
func getPrivacyBehavior(
   for complication: CLKComplication,
   withHandler handler:
   @escaping (CLKComplicationPrivacyBehavior) -> Void) {

    // Hide this complication when locked
    handler(.hidesOnLockScreen)
}
```

Listing 2. Complication privacy behavior.

Watch face templates contain varying *complication* shapes and sizes. Therefore, the data that fits a certain template may not fit other templates. Applications handle each template accordingly allowing users to choose a *complication* on their favorite watch face with meaningful and easily accessible information.

Complication templates, based on their layout and sizes, can display text, image and color. A weather application may display the conventionally associated color for a given temperature on a certain template and the temperature in numbers on another template.

The interface objects for *complications* are provided by the system. Unlike most interface objects in iOS and watchOS, ClockKit framework uses provider classes such as `CLKTextProvider`, `CLKImageProvider`, `CLKFullColorImageProvider`, i.e., for text, image, color and other supported content types. An example of an implementation can be seen in the Appendix, Figure 7 shows the output preview in the Xcode Simulator.

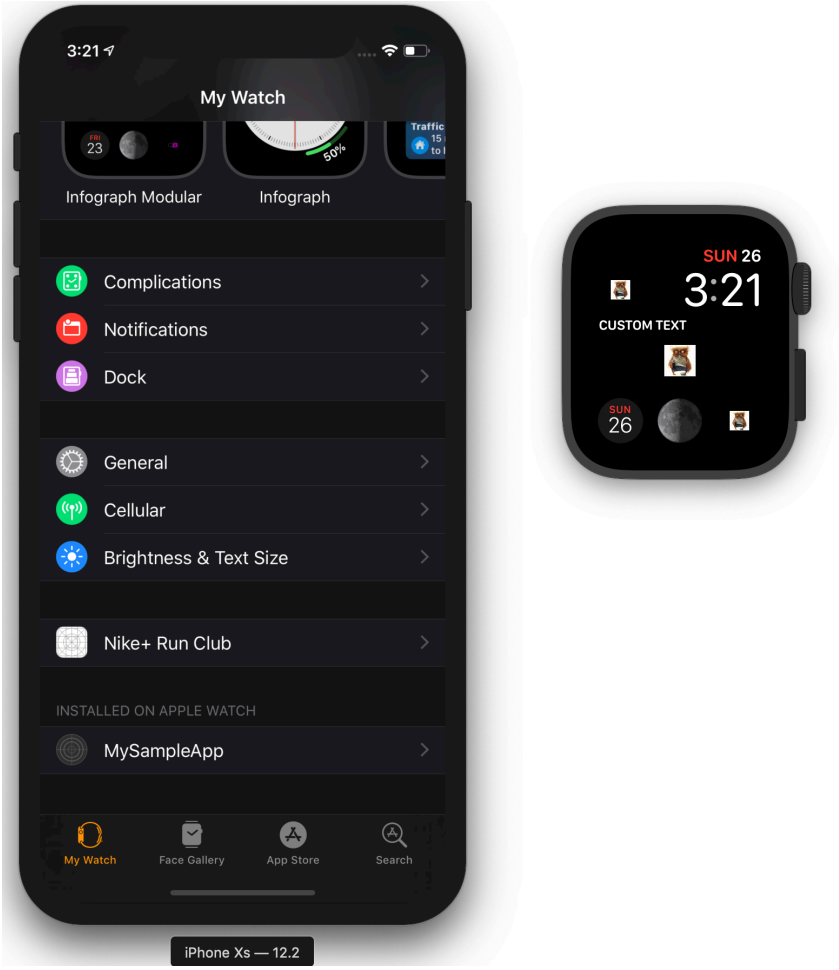

Figure 7. Modular watch face in Simulator

The developer is required to implement two methods: `getCurrentTimelineEntry` and `getSupportedTimeTravelDirections`. The first method is used to provide *complication* with information needed to be presented at this exact moment, the second one defines if the *complications* can show future and past entries. Applications available in several languages would implement helper method `getLocalizableSampleTemplate` to provide localized data for users. [17].

From provided past, current and future entries ClockKit creates a timeline to populate templates for *complications* as shown in Figure 8. This insures that at specified time and date the displayed information will be relevant to the user (see Figure 9). [15].

More accurately, it will show the information which developer thought would be relevant to the user. The responsibility to provide appropriate data lies on the developer.

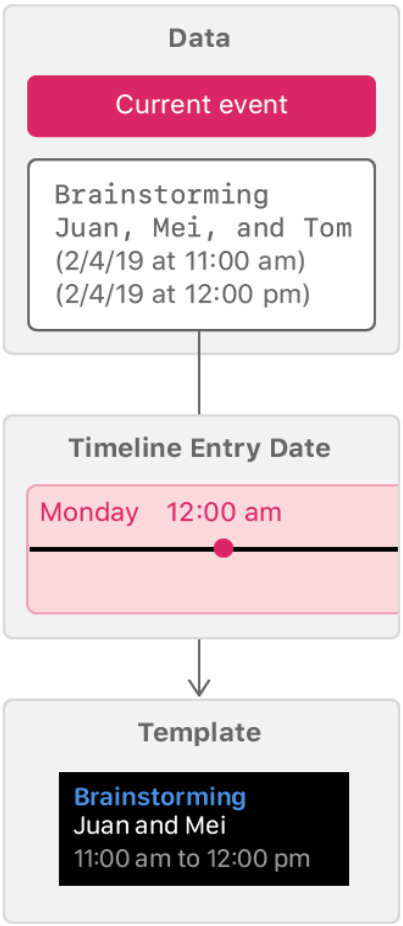

Figure 8. The app data, timeline entry date, and template for a timeline entry [18].

Figure 9. Example of a timeline for modular large complication [14].

3.3 Watch Application

Adding a watch extension to an iOS app involves creating two bundles, Watch App bundle for UI (storyboard and resources such as images) and WatchKit Extension bundle for user interaction logic [19]. This can be compared to having a single bundle for an iOS application's storyboard and assets and another bundle to contain the code responsible for managing storyboard views and asset resources.

The first version of watchOS contained the WatchKit Extension bundle inside the iOS device. As a result, the Watch App was run on the same platform as the iOS application. [21].

The next major update to watchOS 2 brought an autonomous new design and architecture to the watch platform. Sharing platform SDK with the iOS application became a thing of the past and the watch was able to contain its resources and application source code in its own hardware. [21, 22].

During installation of an iOS App, the Watch App bundle containing the WatchKit Extension is downloaded to the iOS device (see Figure 10). These bundles are then copied to the watch hardware during first time pairing. The bundles will remain stored both in the iOS and the watch hardware allowing reinstallation when the watch is cleared or rested. [21, 22].

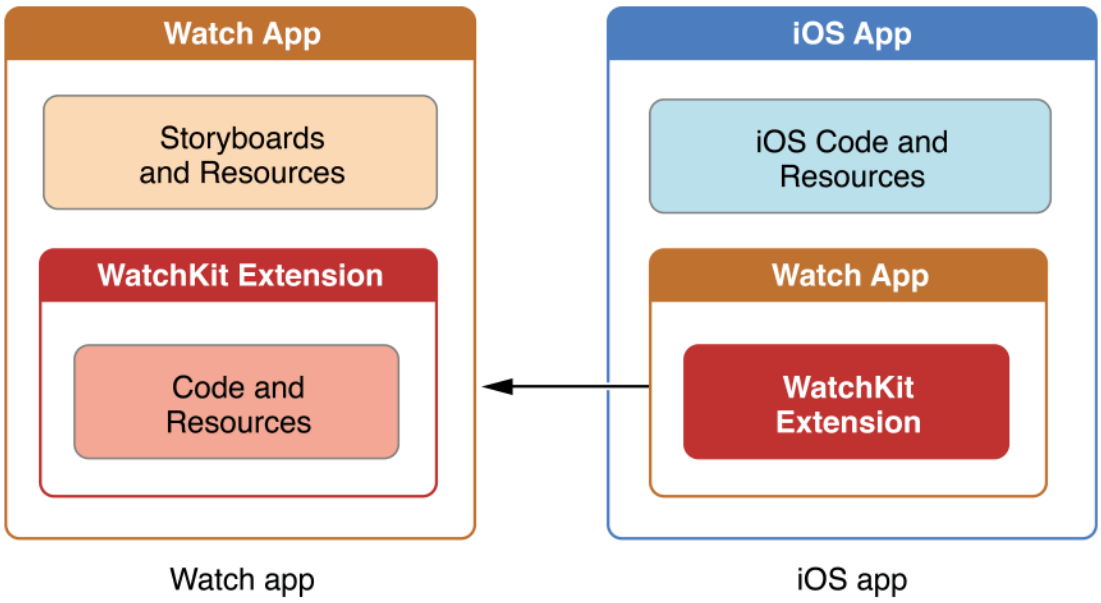

Figure 10. Watch App in iPhone and Apple Watch hardware [19].

In addition to source code, WatchKit Extension can also contain resources such as images, media files and custom fonts. Consequently, developers need to pay attention when fetching resources as a look up at the wrong bundle will simply fail. Listing 3 shows fetching an image named “Image 1” from WatchKit App bundle and WatchKit Extension bundle respectively to assign as property of the interface object `imageObject` (an instance of WKInterfaceImage). [21, 22].

```
// Fetches WatchKit App bundle
imageObject.setImageNamed("Image 1")

// Fetches current bundle, i.e. WatchKit Extension bundle
// Note: Source code resides in WatchKit Extension bundle
imageObject.setImage(UIImage(named: "Image 1")
```

Listing 3. Fetch resource from WatchKit App and WatchKit Extension bundles

With the release of watchOS 2, Apple introduced a separate platform SDK for watch apps with its particular architecture. Frameworks inside the iOS application were no longer accessible in the watch application and interface objects unique to watchOS were needed to build a watch app interface (see Table 1). In watchOS 1, it was possible to dynamically create interface objects in code (in the iOS platform) and send a snapshot of these views to the watch. Following the update to watchOS 2, all interface objects need to be created statically inside storyboard of the Watch App bundle. [21, 22].

Interface Object	Classes	Function	watchOS
Label	WKInterfaceLabel	Displays text	2.0+
Button	WKButtonInterface	Button control	2.0+
Image	WKInterfaceImage	Displays image	2.0+
Audio & Video	WKInterfaceMovie, WKInterfaceInlineMovie	Plays audio and video contents	2.0+, 3.0+
Picker	WKInterfacePicker	Control for list of options	2.0+
Slider	WKInterfaceSlider	Control for floating-point range	2.0+
Table	WKInterfaceTable	Displays table of content	2.0+
Context Menus	WKInterfaceMenu	Presents actions	2.0+
Alerts & Action sheets	Functional	Presents error or prompt interface	2.0+

Table 1. List of interface objects

3.3.1 WatchKit App bundle

The Watch App bundle houses storyboard of watchOS interfaces inside the watch's hardware. This does not include any code, instead Xcode generates binary which is inserted to the bundle at build time and later copied to the watch during installation. [19, 21, 22].

Similar to any bundle in Xcode ecosystem, the watch app bundle must have a unique identifier, an associated property list (info.plist) in addition to being code signed. Failure to correctly configure each bundle requirements will very likely result in unsuccessful installation and the specific error is logged on the paired iPhone's console. Xcode generates default configurations enough to get started with development. [21, 22].

Storyboard contains the app's interface navigation (required to support multiple interfaces) along with scenes that are presented on the watch screen. Scenes can be compared to view controllers on iOS and can have different layout and appearances for different watch sizes similar to size classes in iOS storyboard. Each scene is presented and dismissed by the interface navigation according to user's interaction or possibly system event. [21, 22].

Unlike iOS storyboard, watchOS storyboard understandably restricts layout of interface objects to appear in a vertical stack (see Figure 11). The watchOS will also tweak or fine tune the appearance of interface objects according to the section of the screen they are currently displayed on. [21, 22].

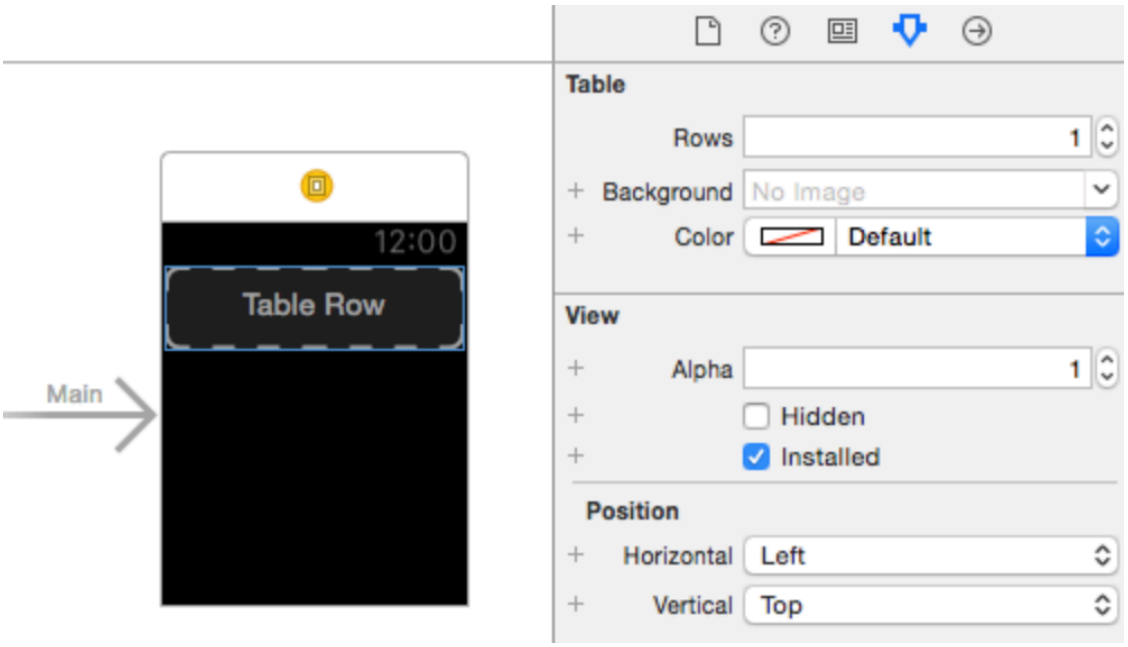

Figure 11. WatchOS storyboard in Xcode

Efficient memory usage is critical on watchOS. Interface objects are loaded into memory only when necessary for displaying. [21].

Interface objects' layout inside a given scene is managed by the vertical stack Xcode generates. However, one can achieve fitting views horizontally or vertically using Group objects (WKInterfaceGroup) whose mere purpose is arranging child interface objects along either horizontal or vertical axis (see Figure 12). Groups can be nested and configured with different insets and spacing values ensuing high customization. [21, 22].

Groups on watchOS bear resemblance to stack view (UIStackView) component in iOS which was introduced in 2015 for iOS9+ devices. Like stack view, Groups are not drawn on screen and are used exclusively to layout their child components. However, unlike stack views, groups support adding a background color, background image and corner radius to the container. The latest watchOS 4 supports an additional overlap layout for grouped components. [21, 22].

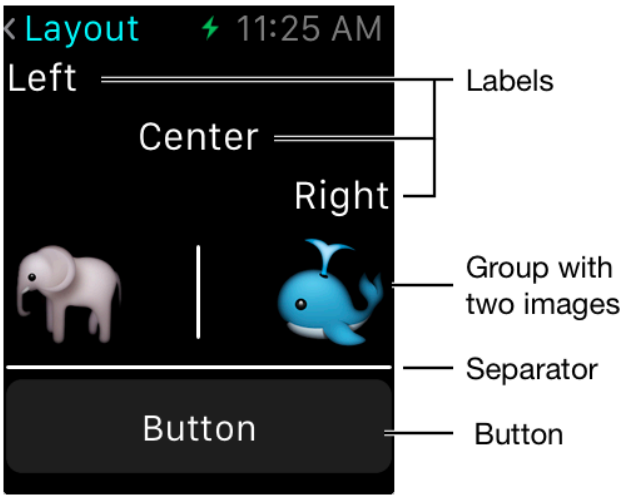

Figure 12. Interface objects in Xcode watch simulator

3.3.2 WatchKit Extension bundle

The WatchKit Extension bundle is where the watch application source code lives. This bundle has a strong dependency on the Watch App bundle and is contained within the Watch App bundle. Interface objects created inside storyboard (inside the Watch App bundle) are referenced in the Extension's interface controller classes.

In watchOS 2, the Apple Watch team introduced WKExtensionDelegate, a protocol defining the application's lifecycle (e.g. application is launched or dismissed) and app wide events (e.g. audio currently playing in the iOS device). An application may choose what app events are of interest and handle appropriately. Table 2 shows the common lifecycle interfaces between iOS UIApplicationDelegate and watchOS WKExtensionDelegate. [19, 21, 22].

Lifecycle events	UIApplicationDelegate & WKExtensionDelegate Methods
Not running – Active	applicationDidFinishLaunching applicationDidBecomeActive
Background – Foreground	applicationWillEnterForeground applicationWillResignActive applicationDidEnterBackground
Device orientation change	application:willChangeStatusBarOrientation (iOS) deviceOrientationDidChange (watchOS)

Table 2. Common lifecycle interfaces of iOS and watchOS

In addition to the common lifecycle events, WKExtensionDelegate brings events unique to watchOS for handling actions that initiated from paired iOS device or Siri through SiriKit. [21, 22].

These include:

- Handling SiriKit intent, new API for watchOS 5.0+
- Audio playback events on paired iOS device, new API for watchOS 5.0+
- Workout session event on paired iOS device (e.g. workout started)
- Recover from interrupted workout session, new API for watchOS 5.0+

The extension can support accessing standard system URLs in order to make phone calls, send SMS or other services. The pattern used to access system URLs from the watch extension is quite similar to the iOS correlated implementation. The extension delegate (WKExtensionDelegate) is a property of WKExtension shared object as is the case with UIApplication and UIApplicationDelegate in iOS. The shared instance of WKExtension allows accessing these standard system URLs via `openSystemURL` API. [21, 22].

There are four main roles the extension implements to enhance user experience:

- Application: The general user interface and navigation when user opens the Watch App.
- Notifications: Handling received notifications, static and dynamic information.
- Glance: Handle single page glance. For example, user viewing a snapshot of the latest state of the application without opening the Watch App.
- *Complications*: Continuous and up to date information via *complications* interface. For example, a weather application should periodically update its *complication* interface to provide meaningful information.

For each role, WatchKit Extension provides supplementary controllers. The extension should implement the associated actions. The following source code shows a simple Watch application that displays an image (see Listing 4). [21, 22].

```
class InterfaceController: WKInterfaceController {

    @IBOutlet weak var imageInterface: WKInterfaceImage!

    override func willActivate() {
        // This method is called when watch view controller
        // is about to be visible to user
        super.willActivate()

        imageInterface.setImage(UIImage(named: "AppIcon"))
    }

}
```

Listing 4. Interface controller that displays an image

4 Data and resources

Watch app extension supports displaying variety of data to the user. Text, image, audio and video are all displayable on the user's wrist. Static data can be fetched from local storages of the WatchKit App bundle and the Extension bundle. In addition, data is stored in the documents and caches folders. [23].

4.1 Sharing between bundles

Sharing data between the WatchKit App and Extension bundles is often necessary. Resources that ship with the application such as custom fonts or media files sometimes need to be set in both bundles. For example, an image can be set on an interface object (WKInterfaceImage) in storyboard inside the Watch App bundle and the same image could be accessed in the source code inside WatchKit Extension. It is recommended to follow one of the following techniques for efficient data sharing [23]:

- Duplicate the resource in both bundles. Placing copies of the same resources inside both bundles allows each bundle to access its own copy. This is recommended for small files such as fonts.

- Place resources that are dynamically accessed at runtime inside WatchKit Extension bundle. If these resources are accessed also inside the Watch App bundle, the bundle is capable of fetching WatchKit Extension bundle automatically. This is because the WatchKit Extension is placed inside Watch App bundle. For example, audio and video files that ship with the app should follow this technique. The source code will be able to load and play the media content using APIs provided inside the NSBundle class.

- Use a common shared container for media files that are acquired at runtime. These include files that are downloaded from the network or transferred from the iOS application.

The majority of resources are dynamically obtained and stored. Therefore, the last technique of sharing data between Watch App bundle and WatchKit Extension bundle using shared containers is the most common. Creating this shared container can be done by enabling the feature in Xcode capabilities configuration and obtaining an App Group bundle identifier (see Figure 13). The App Group must be enabled in both the Watch App and WatchKit Extension targets. [23].

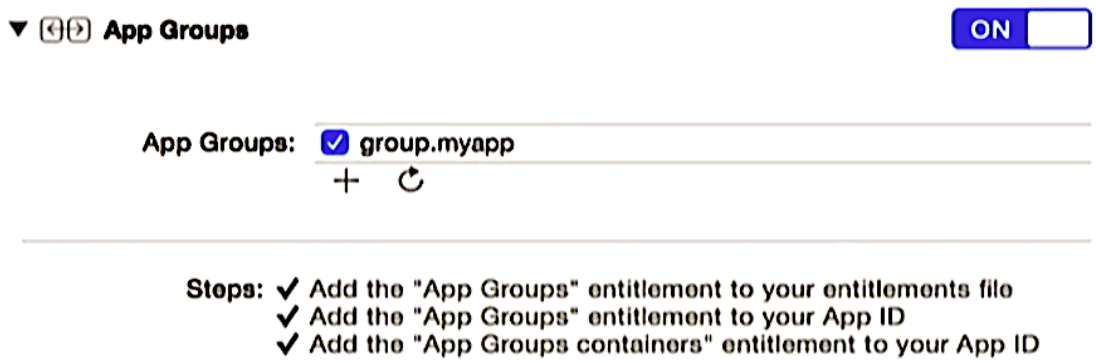

Figure 13. App Groups settings in Xcode

The following source code demonstrates accessing an audio file from a shared container with bundle identifier "SampleApp.myapp" (see Listing 5).

```
func audioFileURL(named name: String) -> URL? {
    let fileManager = FileManager.default

    guard let container = fileManager
        .containerURL(forSecurityApplicationGroupIdentifier:
        "SampleApp.myapp"),
        let filename = (name as NSString)
        .appendingPathExtension("mp4") else {
            return nil
    }

    let audioFileURL = container.appendingPathComponent(filename)

    return audioFileURL
}
```

Listing 5. Accessing audio file

App Group, the shared container, provides secure access from multiple processes. There is no file management required for the shared container however it is required to wait for data write operations by the extension before performing any read operations. [23].

4.2 Sharing with iOS app, Connectivity

In watchOS 2, Apple introduced Watch Connectivity framework that supports communication with associated iOS app. The connection can take place while both apps are in foreground, when only the sender (the app initiating the connection) is in foreground or even when both apps are in the background. In the latter case, the receiver

app will not be informed of the received data up until it is brought to the foreground. [23 - 26].

Connectivity framework supports different modes of file transfer suited for given data size and required urgency for synchronization. Recommended data sharing technics are discussed below. [23 - 26].

4.2.1 Application state updates

Application state updates are sequential context updates used to synchronize the iOS and watchOS apps. It is an asynchronous file transfer that is started by the system. Triggering a file transfer in this mode simply adds the request to the system's internal queue to be executed when possible. The communication can be used to update interface to the latest context. [23 - 26].

This is used for small amounts of state information and only the last received update is used at the receiver side i.e., latest data replaces old data. These transfers may happen in the background when both apps are inactive. For example, a music app can use this mode of file transfer to update title and artwork of the currently playing audio to synchronize the iOS and watchOS applications. [23 - 26].

The following Listing 6 shows sending an update of a new music track title to the Watch App.

```
import WatchConnectivity

// New music track title currently playing.
// Need to notify Watch App
@available(iOS 9.3, *)
func didSelectTrack(titled newTitle: String) {

    let session = WCSession.default

    // Check if there is a paired Watch App
    guard session.isPaired, session.isWatchAppInstalled else {
        return
    }

    // Application context is always a key value
    // Watch App must use the same key to access
    let new = ["trackTitle" : newTitle]

    // Update Watch session
```

```
    try? session.updateApplicationContext(new)
}
```

Listing 6. Update application context from iOS app

The Watch App receives application context updates via its WCSessionDelegate. The implementation should extract the update and reflect by applying the changes to the watch interface as shown in Listing 7. [26].

```
@IBOutlet var trackTitleLabel: WKInterfaceLabel!

// MARK: WCSessionDelegate implementation

public func session(
    _ session: WCSession,
    didReceiveApplicationContext
    applicationContext: [String : Any]) {

    guard let newTrackTitle =
        applicationContext["trackTitle"] as? String else {
        // This is not track title update
        return
    }

    DispatchQueue.main.async { [weak self] in
        self?.trackTitleLabel.setText(newTrackTitle)
    }
}
```

Listing 7. Receiving application context update on Watch App

4.2.2 Complication updates

Complication updates use a high-priority communication mode for its key-value pair data transfers. It is a single direction data transfer to update *complications* of the Watch App using latest data in the iOS app. Receiving this type of updates may activate the Watch App if it was not active. WatchOS allows a limited number of these high-priority data transfers in the period of 24 hours. Developers are responsible for managing the most efficient time for such data transfers. The WCSession API to transfer the key-value pair data is “transferCurrentComplicationUserInfo”. If invoked more times than allowed, the transfer will take place via the less prioritized transferUserInfo API discussed below. [23 - 27]. Listing 8 shows sending these updates.

```
guard session.activationState == .activated else {
```

```
    // Activate and observe the state change
    // Send only if state is .activated
    return
}

// Complication update.
// New update for complication that displays date
let new = ["complicationDate" : newDate]

// Can only be called limited times a day
session.transferCurrentComplicationUserInfo(new)
```

Listing 8. Complication data sharing.

4.2.3 Background data transfer

Connectivity framework provides There is no invoked limit on this mode of communication. It can be triggered using WCSession method `transferUserInfo` as many times as necessary. [24 - 26].

When transferring key-value pair data, it is required to ensure that the session is activated before initiating the transfer. This applies for both *complication* and arbitrary key-value pair data transfer as shown in Listing 9. [24 - 26].

```
// New message to the Watch or iOS device
let message = ["messageKey" : messageData]

// Can be called as many times as necessary
session.transferUserInfo(message)
```

Listing 9. Message data sharing.

4.2.4 File transfer

Files such as images can be transferred via the `transferFile` API of WCSession. The system determines the appropriate time to initiate this type of transfers considering power level and other application status. The receiver is expected to handle the received file immediately. Unhandled received file is deleted. [21, 22, 24 - 26]. (see Listing 10)

```
// Queue a file transfer
// Optionally send metadata along with the file
session.transferFile(fileURL, metadata: nil)
```

Listing 10. File sharing.

4.2.5 Immediate arbitrary data transfer

Connectivity framework also supports asynchronous immediate data transfer to a paired device when both applications are active. Technically, the iOS application is considered always active as sending such message from the Watch App causes the iOS app to wake up. The reverse does not hold true and error handler will be invoked if an immediate message is sent to the Watch App while its inactive. [21, 22, 24 - 26]. Sharing of key-value pair data is shown in Listing 11.

```
guard session.activationState == .activated else {
    // Activate and observe the state change
    // Send only if state is .activated
    return
}

// Complication update.
// Can only be called limited times a day
session.transferCurrentComplicationUserInfo(new)

// OR, For non-complication updates

// Any key value update.
// Can be called as many times as necessary
session.transferUserInfo(new)
```

Listing 11. Key-value pair data sharing

All key-value data related updates, including *complication* updates, are received via the same `WCSessionDelegate` API as shown in Listing 12.

```
// Key-Value pair data received
func session(_ session: WCSession,didReceiveUserInfo userInfo:
[String : Any] = [:]) {
    let cKey = "The complication key"

    guard let complicationUpdate = userInfo[cKey] else {
        // Proccess non-complication update
        return
    }

    let complications = CLKComplicationServer
                        .sharedInstance().activeComplications

    // Invalidate complication data sources
    complications.forEach(server.reloadTimeline(for:))
}

// File received
func session(_ session: WCSession,
             didReceive file: WCSessionFile) {
```

```
    // Receive an image file
    guard let imageData = try? Data(contentsOf: file.fileURL),
        let image = UIImage(data: imageData) else {
            // It was not an image file
            return
    }

    // Set the image on an image interface
    DispatchQueue.main.async { [weak self] in
        self?.imageInterface.setImage(image)
    }
}
```

Listing 12. Receiving transferred data

Upon receiving files, the Watch App can store them inside its Document or Caches folder. The Document folder is non-purgeable directory therefore suitable for critical files. Caches is purgeable and, therefore, the system can free up the cache when in need of more memory. [24 - 26].

4.3 Secure storage, Keychain

Like iOS, watchOS applications contain secure storage. Apple Watch users expect to automatically reply to oncoming messages, open their paired macOS and iOS devices and perform other secure tasks without entering passwords as long as the watch is on their wrist. This is possible because applications can read from and write to Keychain (a secure database library) within Watch Apps. [21, 22, 28].

WatchOS internally determines whether Keychain access should be granted or denied. Accesses when the watch is not on wrist are usually denied. Developers can use this setting by applying the `kSecAttrAccessibleWhenUnlocked` attribute. [28].

The following code snippet, Listing 13, demonstrates storing data into Keychain with access on unlocked attribute.

```
let password = "My password"

if let passwordData
= password.dataUsingEncoding(NSUnicodeStringEncoding) {

    let attributes: [NSString: NSObject] = [
        kSecClass: kSecClassGenericPassword,
        kSecAttrAccessible: kSecAttrAccessibleWhenUnlocked,
```

```
        kSecAttrService: "myservice",
        kSecAttrAccount: "account name",
        SecValueData: password
    ]

    SecItemAdd(attributes, nil)
}
```

Listing 13. Storing to Keychain [28].

5 Networking

The Watch App has access to HTTP and HTTPS connections using NSURLSession class similar to an iOS app. When downloading files in the Watch App, it is best to use a background session [29]. The following Listing 14 shows a sample downloader implementation.

```
class Downloader: NSObject, URLSessionDownloadDelegate  {

    private lazy var downloadSession: URLSession = {
        let config = URLSessionConfiguration
                        .background(withIdentifier: "theID")

        return URLSession(configuration: config, delegate: self,
                          delegateQueue: nil)
    }()

    // Download API
    func download(from url: URL) {
        downloadSession.downloadTask(with: url).resume()
    }

    // MARK: - URLSessionDownloadDelegate

    func urlSession(_ session: URLSession,
                    downloadTask: URLSessionDownloadTask,
                    didFinishDownloadingTo location: URL) {

        // Use the downloaded file before its removed
    }
}
```

Listing 14. A sample downloader class

6 Upcoming features

Every year in summer time Apple holds Worldwide Developers Conference (WWDC) where upcoming changes to software and devices are introduced. Some of the key changes are usually known in advance. WWDC 2019 is not an exception. [30].

Unsurprisingly, Apple is working on giving Watches more autonomy. Apple Watch Series 4 got a model with embedded SIM card, abbreviated eSim, and now a year later the company is introducing a separate App Store, so users can download applications directly from their wrists. Currently, the only way to add application to the wearable device is from companion “Watch” app installed on iOS device. [30].

Having a separate application store could ease the development process hence more software engineers would be attracted to work with Apple Watch. At the moment it is not possible to create a dedicated Watch App without an existing iOS application. [30].

Other known changes to come include several popular Apple applications to get an alter on the wearables: iBooks, Voice Memos, Calculator. Furthermore, there will be enhancements in health tracking as usual. [30].

User interface will be upgraded with new watch faces and additional *complications*. Finally, entertaining Animojis and Memojis will become available on watchOS as well. [30].

7 Conclusion

The purpose of this thesis is to present the key features and capabilities of Apple Watch and the challenges of application development. Apple Watch applications’ relationship with their counterpart iOS applications and the communication principles between the two platforms are discussed and demonstrated.

Apple Watch applications require similar development process. The Watch platform is designed as a supplement to iOS applications. Therefore, it is critical to form a synchronized information flow with the sibling platform applications.

Apple team recognizes the limited time that users spend interacting with smartwatch applications. Accordingly, the application development process for Apple Watch focuses on offering the latest and most important information at a glance. The platform's operating system has evolved to support this goal over the years.

The thesis concludes that Apple Watch applications are an important part of the Apple ecosystem. Although they cannot exist independently from a counterpart iOS application, their interconnectivity results in unparalleled convenient user experience. The development process takes place in both the Watch and iOS device platforms. A well-designed Watch application can fulfill most tasks of the iOS application with quick on hand interactions.

Apple Watch is one of the latest product categories for the company. The fast development and improvements on the platform, in addition to the fast-growing user-base, increases the necessity of Watch App extensions to iOS applications. Progressive companies and businesses take this opportunity to offer a better and more integrated user experience for those using Apple Watch.

References

1. Edward Roberts, William Yates. Altair 8800 Computer. Popular Electronics, January 1975 [online]. URL: http://www.swtpc.com/mholley/PopularElectronics/Jan1975/PE_Jan1975.htm Accessed May 29, 2019

2. Bob Walz, Rebecca Knesel. Motorola demonstrates portable telephone to be available for public use by 1976 [online]. Motorola Inc. April 3, 1973 URL: motorola.com/sites/default/files/library/us/about-motorola-history-milestones/pdfs/DynaTAC_newsrelease_73_001.pdf Accessed May 29, 2019

3. Chance Miller. Strategy Analytics: Apple Watch accounted for half of all smartwatch sales in 2018 [online]. February 27, 2019 URL: 9to5mac.com/2019/02/27/apple-watch-sales-2018 Accessed May 29, 2019

4. Joe Thompson. Four Revolutions. The lost chapter: a concise history of the LED [online]. February 26, 2018 URL: hodinkee.com/articles/four-revolutions-led-watches Accessed May 29, 2019

5. History of Smart Watch [online]. Mount Tech. November 4, 2016 URL: mount-tech.blogspot.com/2016/11/history-of-smart-watch.html Accessed May 29, 2019

6. Nat Kerris, Sarah O'Brien. Apple Unveils Apple Watch - Apple's Most Personal Device Ever [online]. Apple Inc. September 9, 2014 URL: apple.com/newsroom/2014/09/09Apple-Unveils-Apple-Watch-Apples-Most-Personal-Device-Ever Accessed May 29, 2019

7. Amy Bessette. Apple introduces Apple Watch Series 2, the ultimate device for a healthy life [online]. Apple Inc. September 7, 2016 URL: apple.com/newsroom/2016/09/apple-introduces-apple-watch-series-2 Accessed May 29, 2019

8. Lance Lin. Apple Watch Series 3 brings built-in cellular, powerful new health and fitness enhancements [online]. Apple Inc. September 12, 2017 URL: apple.com/newsroom/2017/09/apple-watch-series-3-features-built-in-cellular-and-more Accessed May 29, 2019

9. Lance Lin. Apple Watch Series 4: Beautifully redesigned with breakthrough communication, fitness and health capabilities [online]. Apple Inc. September 12, 2018 URL: apple.com/newsroom/2018/09/redesigned-apple-watch-series-4-revolutionizes-communication-fitness-and-health Accessed May 29, 2019

10. Notifications on your Apple Watch [online]. Apple Inc. October 16, 2018 URL: https://support.apple.com/en-ug/HT204791 Accessed May 29, 2019

11. Human Interface Guidelines. System Capabilities. Notifications [online]. Apple Inc. 2019 URL: developer.apple.com/design/human-interface-guidelines/watchos/system-capabilities/notifications Accessed May 29, 2019

12. App Programming Guide for watchOS. Overview. Developing for Apple Watch [online]. Apple Inc. December 12, 2016 developer.apple.com/library/archive/documentation/General/Conceptual/WatchKitProgrammingGuide Accessed May 29, 2019

13. Human Interface Guidelines. App Architecture. Complications [online]. Apple Inc. 2019 URL: developer.apple.com/design/human-interface-guidelines/watchos/app-architecture/complications Accessed May 29, 2019

14. Creating Complications with ClockKit [online]. WWDC 2015 Session 209 watchOS. Apple Inc. URL: developer.apple.com/videos/play/wwdc2015/209 Accessed May 29, 2019

15. App Programming Guide for watchOS. Complications. Complication Essentials [online]. Apple Inc. December 12, 2016 URL: developer.apple.com/library/archive/documentation/General/Conceptual/WatchKitProgrammingGuide/ComplicationEssentials.html Accessed May 29, 2019

16. Developing Complications for Apple Watch Series 4 [online]. Tech Talks Session 208 watchOS. Apple Inc. developer.apple.com/videos/play/tech-talks/208/ Accessed May 29, 2019

17. Documentation. ClockKit. CLKComplicationDataSource [online]. Apple Inc. 2019 URL: developer.apple.com/documentation/clockkit/clkcomplicationdatasource Accessed May 29, 2019

18. Documentation. WatchKit. Creating an Effective watchOS Experience. Adding a Complication to Your watchOS App. Providing Data for Your Complication. Creating a Timeline Entry [online]. Apple Inc. 2019 URL: developer.apple.com/documentation/clockkit/adding_a_complication_to_your_watchos_app/providing_data_for_your_complication/creating_a_timeline_entry Accessed May 29, 2019

19. Documentation. WatchKit. Creating an Effective watchOS Experience. Setting Up a watchOS Project [online]. Apple Inc. 2019 URL: developer.apple.com/documentation/watchkit/creating_an_effective_watchos_experience/setting_up_a_watchos_project Accessed May 29, 2019

20. App Programming Guide for watchOS. Overview. Configuring Your Xcode Project [online]. Apple Inc. December 12, 2016 URL: developer.apple.com/library/archive/documentation/General/Conceptual/WatchKitProgrammingGuide/ConfiguringYourXcodeProject.html Accessed May 29, 2019

21. WatchKit In-Depth, Part 1. WWDC 2015 Session 207 watchOS [online]. Apple Inc. URL: developer.apple.com/videos/play/wwdc2015/207 Accessed May 29, 2019

22. WatchKit In-Depth, Part 2. WWDC 2015 Session 207 watchOS [online]. Apple Inc. URL: developer.apple.com/videos/play/wwdc2015/208 Accessed May 29, 2019

23. App Programming Guide for watchOS. Overview. Sharing Data [online]. December 12, 2016 URL: developer.apple.com/library/archive/documentation/General/Conceptual/WatchKitProgrammingGuide/SharingData.html Accessed May 29, 2019

24. Introducing Watch Connectivity. WWDC 2015 Session 713 watchOS [online]. Apple Inc. URL: developer.apple.com/videos/play/wwdc2015/713 Accessed May 29, 2019

25. Documentation. WatchConnectivity [online]. Apple Inc. 2019 URL: developer.apple.com/documentation/watchconnectivity Accessed May 29, 2019

26. Documentation. WatchConnectivity. WCSessionDelegate [online]. Apple Inc. 2019 URL: developer.apple.com/documentation/watchconnectivity/wcsessiondelegate Accessed May 29, 2019

27. Documentation. WatchConnectivity. WCSession. transferCurrentComplicationUserInfo [online]. Apple Inc. 2019 URL: developer.apple.com/documentation/watchconnectivity/wcsession/1615639-transfercurrentcomplicationuseri Accessed May 29, 2019

28. App Programming Guide for watchOS. Overview. Leveraging iOS Technologies [online]. Apple Ind. December 12, 2016 URL: developer.apple.com/library/archive/documentation/General/Conceptual/WatchKitProgrammingGuide/iOSSupport.html Accessed May 29, 2019

29. Documentation. WatchKit. Keeping Your watchOS Content Up to Date [online]. Apple Inc. 2019 URL: developer.apple.com/documentation/watchkit/keeping_your_watchos_content_up_to_date Accessed May 29, 2019

30. Mark Gurman. Apple to Reveal New Home-Grown Apps, Software Features at WWDC [online]. Bloomberg. May 6, 2019 URL: www.bnnbloomberg.ca/apple-to-reveal-new-home-grown-apps-software-features-at-wwdc-1.1254184

```
class ComplicationController: NSObject, CLKComplicationDataSource {

    // MARK: - Timeline Configuration

    func getSupportedTimeTravelDirections(
                for complication: CLKComplication,
                withHandler handler: @escaping (CLKComplicationTimeTravelDirections) -> Void) {
        handler([.forward, .backward])
    }

    // MARK: - Timeline Population
    func getCurrentTimelineEntry(
            for complication: CLKComplication,
            withHandler handler: @escaping (CLKComplicationTimelineEntry?) -> Void) {
        // Call the handler with the current timeline entry

        getPlaceholderTemplate(for: complication) { template in
            guard let template = template else {
                handler(nil)
                return
            }
            let entry = CLKComplicationTimelineEntry(date: Date(), complicationTemplate:
template)
            handler(entry)
        }
    }

    // MARK: - Optional Methods

    func getPrivacyBehavior(
            for complication: CLKComplication,
            withHandler handler: @escaping (CLKComplicationPrivacyBehavior) -> Void) {
        handler(.showOnLockScreen)
    }

    // MARK: - Placeholder Templates

    func getLocalizableSampleTemplate(
            for complication: CLKComplication,
            withHandler handler: @escaping (CLKComplicationTemplate?) -> Void) {
        // This method will be called once per supported complication, and the results will be
cached

        getPlaceholderTemplate(for: complication) { template in
            handler(template)
        }
    }
```

```
func getPlaceholderTemplate(
        for complication: CLKComplication,
        withHandler handler: @escaping (CLKComplicationTemplate?) -> Void) {

    switch(complication.family) {

    case .modularSmall, .modularLarge, .graphicRectangular:
      guard let modularImage = UIImage(named: "Complication/Modular") else {
        handler(nil)
        return
      }

      if complication.family == .modularSmall {
        // Construct a template that displays an image and a short line of text.
        let template = CLKComplicationTemplateModularSmallStackImage()

        // Set the data providers.
        template.line1ImageProvider = CLKImageProvider(onePieceImage: modularImage)
        template.line2TextProvider = CLKSimpleTextProvider(text: "text",
                                                           shortText: "shortText")
        handler(template)
        return
      }

      if complication.family == .modularLarge {
        // Construct a template for displaying a header row and a tall row of body text.
        let template = CLKComplicationTemplateModularLargeTallBody()

        // Header text provider cannot be nil
        template.headerTextProvider = CLKTextProvider
                .localizableTextProvider(withStringsFileTextKey: "modularLarge header")

        // Provide for example date for the body
        template.bodyTextProvider =
                CLKDateTextProvider(date: Date(), units: [.weekday, .day, .month])

        handler(template)
        return
      }

      if complication.family == .graphicRectangular {
        // Construct template for displaying a large rectangle
        // containing header text and an image.
        let template = CLKComplicationTemplateGraphicRectangularLargeImage()
        template.imageProvider =
CLKFullColorImageProvider(fullColorImage: modularImage)
```

```
            template.textProvider =
    CLKTextProvider.localizableTextProvider(withStringsFileTextKey: "Custom text")

            handler(template)
            return
          }

        case .circularSmall, .graphicCircular:
          guard let circularImage = UIImage(named: "Complication/Circular") else {
            handler(nil)
            return
          }

          if complication.family == .circularSmall {
            let template = CLKComplicationTemplateCircularSmallRingImage()
            // The image will be a template and in most cases the colors will be overriden
            template.imageProvider = CLKImageProvider(onePieceImage: circularImage)
            handler(template)
          }
          if complication.family == .graphicCircular {
            let template = CLKComplicationTemplateGraphicCircularImage()
            // The image will be rendered with provided colors
            template.imageProvider = CLKFullColorImageProvider(fullColorImage:
circularImage)
            handler(template)
          }

        // Handle any non-supported families.
        default:
          handler(nil)
        }
    }
}
```