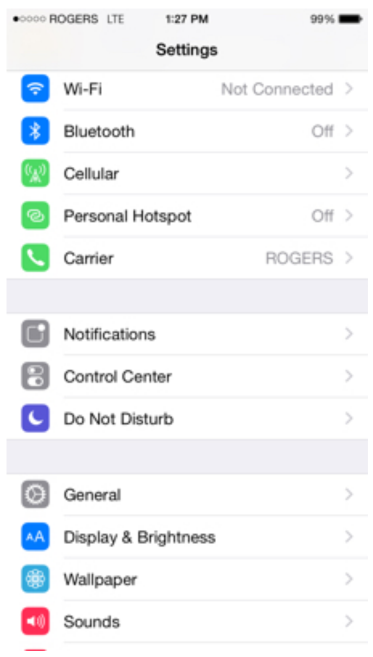
CHAPTER SIXTEEN: PERSISTENCE


●○○○○ ROGERS LTE 1:27 PM 99%

Settings

Wi-Fi Not Connected

Bluetooth Off

Cellular

Personal Hotspot Off

Carrier ROGERS

Notifications

Control Center

Do Not Disturb

General

Display & Brightness

Wallpaper

Sounds

I. DATA SOURCE AND PERSISTENCE

Up to this point, there is no persistence in the application. When we close the app and open it again, all the changes that we have done in the last session are lost. Have you wondered where are the data of an iOS application from? How to make them persist through app stops?

There are many ways to obtain data for an application. Most modern applications get their data from network. They launch HTTP requests to the URL of their backend servers to get data, and the response data is cached locally and displayed in the views. Applications like YouTube, Instagram, FaceBook use very little local data storage and can generate much data traffic because of their HTTP requests.

Data persistence is the mechanism of storing any type of data to disk so that the same data can be retrieved without being altered when the user opens the app next time. While you could save the same data on a server, in this tutorial we are going to describe all the ways to store data locally in iOS apps.

Another type of data source is to use local storage, like the Calculator app. It generate few HTTP requests. Most of its assets are stored locally via CoreData, a persistence and/or in-memory cache framework that consists of a very powerful set of other tools that can be used within the app. In most common examples CoreData is used as a wrapper for the SQLite database and it’s used to save and present any type of user data. When there is no Internet access, or an unstable one, using CoreData can make the app more accessible.

Those are just two of the many ways to achieve data persistence, here is a non-exhaustive list of persistence approaches:

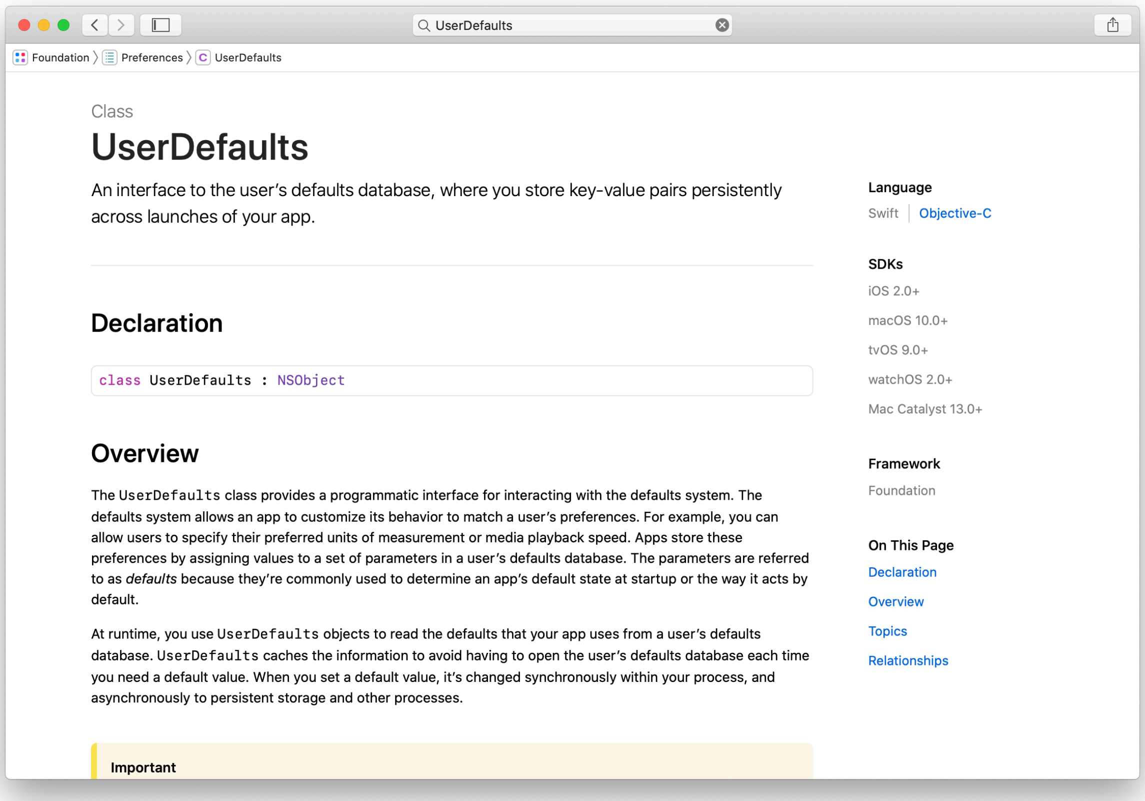
- UserDefaults: often used to save a small amount of data
- Property List: often used to save user settings and preferences
- SQLite: stores with large amount of data with relationships
- KeyChain: saves sensitive data like passwords, credit card information.
- CoreData: Apple's native persistence solution, persists data of any form and retrieve it

All these approaches have different use cases, so when you want to persist some data of your app, you should choose the most suitable way of persistence for your data.

In our PowerFocus application, there is neither backend server nor much data to store. What we need to store are the properties of the timer: lengths of works/breaks/long breaks, break frequency, completed cycles… That information is lightweight and preference-related, so using UserDefaults is the most suitable way.

II. USER DEFAULTS

Like many utilities that we have used, Apple provides a class UserDefaults to help us. Open the Developer Documentation and search "UserDefaults". Study the class and the usage of its methods:



Class

UserDefaults

An interface to the user's defaults database, where you store key-value pairs persistently across launches of your app.

Declaration

```
class UserDefaults : NSObject
```

Overview

The `UserDefaults` class provides a programmatic interface for interacting with the defaults system. The defaults system allows an app to customize its behavior to match a user's preferences. For example, you can allow users to specify their preferred units of measurement or media playback speed. Apps store these preferences by assigning values to a set of parameters in a user's defaults database. The parameters are referred to as *defaults* because they're commonly used to determine an app's default state at startup or the way it acts by default.

At runtime, you use `UserDefaults` objects to read the defaults that your app uses from a user's defaults database. `UserDefaults` caches the information to avoid having to open the user's defaults database each time you need a default value. When you set a default value, it's changed synchronously within your process, and asynchronously to persistent storage and other processes.

Important

Language
Swift | Objective-C

SDKs
iOS 2.0+
macOS 10.0+
tvOS 9.0+
watchOS 2.0+
Mac Catalyst 13.0+

Framework
Foundation

On This Page
Declaration
Overview
Topics
Relationships

An interface to the user's defaults database, where you store key-value pairs persistently across launches of your app.

The UserDefaults class provides a programmatic interface for interacting with the defaults system. The defaults system allows an app to customize its behavior to match a user's preferences. For example, you can allow users to specify their preferred units of measurement or media playback speed. Apps store these preferences by assigning values to a set of parameters in a user's defaults database. The parameters are referred to as *defaults* because they're commonly used to determine an app's default state at startup or the way it acts by default.

At runtime, you use UserDefaults objects to read the defaults that your app uses from a user's defaults database. UserDefaults caches the information to avoid having to open the user's defaults database each time you need a default value. When you set a default value, it's changed synchronously within your process, and asynchronously to persistent storage and other processes.

To put information in UserDefaults, we need to get the standard user default object. In the class ViewController, add a user default attribute to the view controller:

```
var userDefaults: UserDefaults = UserDefaults.standard
```

In the method viewDidLoad, remove the hard-coded focusTimer constructor and add a method call of initFocusTimer. Create the method initFocusTimer that initializes the focusTimer attribute:

```
override func viewDidLoad() {
    super.viewDidLoad()
    // Do any additional setup after loading the view, typically from a nib.
    self.navigationController?.setNavigationBarHidden(true, animated: false)
    initFocusTimer()
    . . .
}


private func initFocusTimer() -> Void {
    focusTimer = FocusTimer(3, 1, 2, 3, true)
}
```

It is very easy to get or set values from the key-value pairs of user defaults. Simply use these methods:

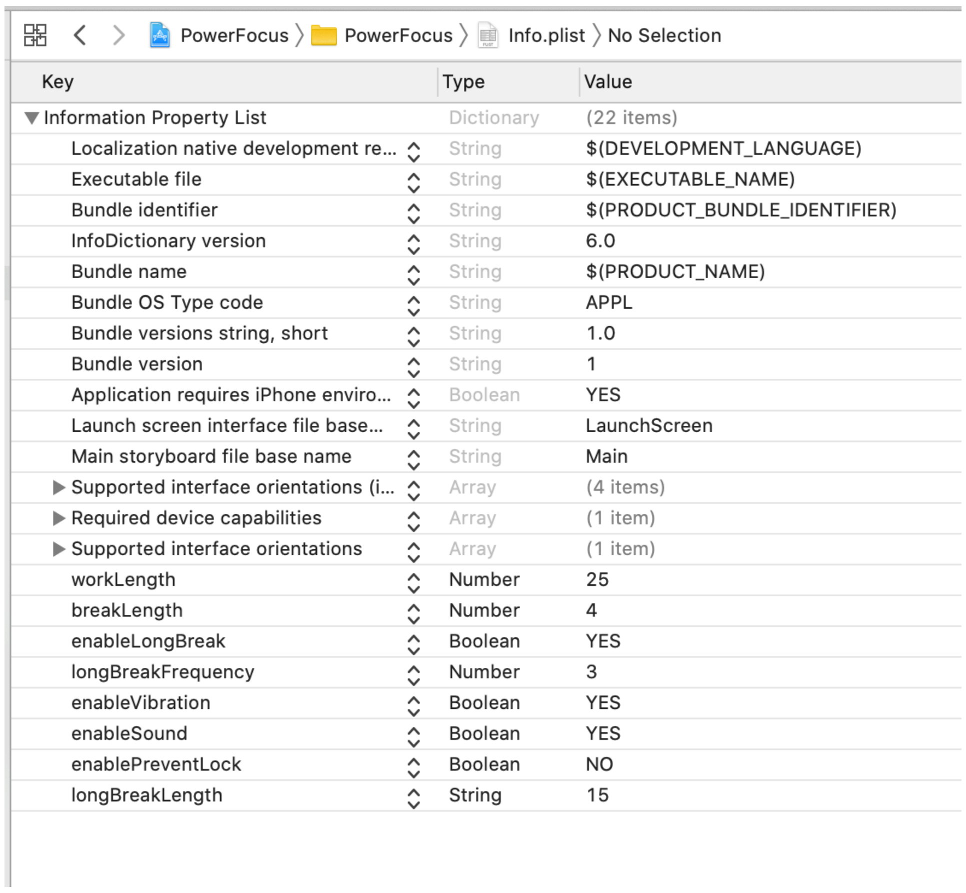
```
func object(forKey: String) -> Any?
        Returns the object associated with the specified key.
func set(Any?, forKey: String)
        Sets the value of the specified default key.
```

When the application is launched for the first time, there aren't any customized user default entries. So we need to see if there are already our entries in the user default dictionary. When there isn't a key in the user default, calling UserDefaults.object(forKey: String) gives nil. So if it's the first time that the view controller is launched, we need to create the entires and put them into user defaults. If there are already some entries, we only need to create a FocusTimer according to the user default. In this case we have to use the getters of user defaults.

Here is the initFocusTimer with the userDefaults' getters and setters:

```
func initFocusTimer() -> Void {
    focusTimer = FocusTimer(3, 1, 2, 3, true)
    if userDefaults.object(forKey: "workLength") == nil {
        // Set default timer attributes for the first launch
        print("Keys not found in user defaults")
        userDefaults.set(25, forKey: "workLength")
        userDefaults.set(4, forKey: "breakLength")
        userDefaults.set(15, forKey: "longBreakLength")
        userDefaults.set(3, forKey: "longBreakFrequency")
        userDefaults.set(true, forKey: "enableLongBreak")
    } else {
        focusTimer = FocusTimer(userDefaults.integer(forKey: "workLength"),
            userDefaults.integer(forKey: "breakLength"),
            userDefaults.integer(forKey: "longBreakLength"),
            userDefaults.integer(forKey: "longBreakFrequency"),
            userDefaults.bool(forKey: "enableLongBreak"))
    }
}
```

Add these entries to the Info.plist. Select Info.plist in the project navigator and right click then choose add row to add the rows of our entires.


PowerFocus 〉 PowerFocus 〉 Info.plist 〉 No Selection

Key	Type	Value
▼ Information Property List	Dictionary	(22 items)
Localization native development re...	String	$(DEVELOPMENT_LANGUAGE)
Executable file	String	$(EXECUTABLE_NAME)
Bundle identifier	String	$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0
Bundle name	String	$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle version	String	1
Application requires iPhone enviro...	Boolean	YES
Launch screen interface file base...	String	LaunchScreen
Main storyboard file base name	String	Main
▶ Supported interface orientations (i...	Array	(4 items)
▶ Required device capabilities	Array	(1 item)
▶ Supported interface orientations	Array	(1 item)
workLength	Number	25
breakLength	Number	4
enableLongBreak	Boolean	YES
longBreakFrequency	Number	3
enableVibration	Boolean	YES
enableSound	Boolean	YES
enablePreventLock	Boolean	NO
longBreakLength	String	15

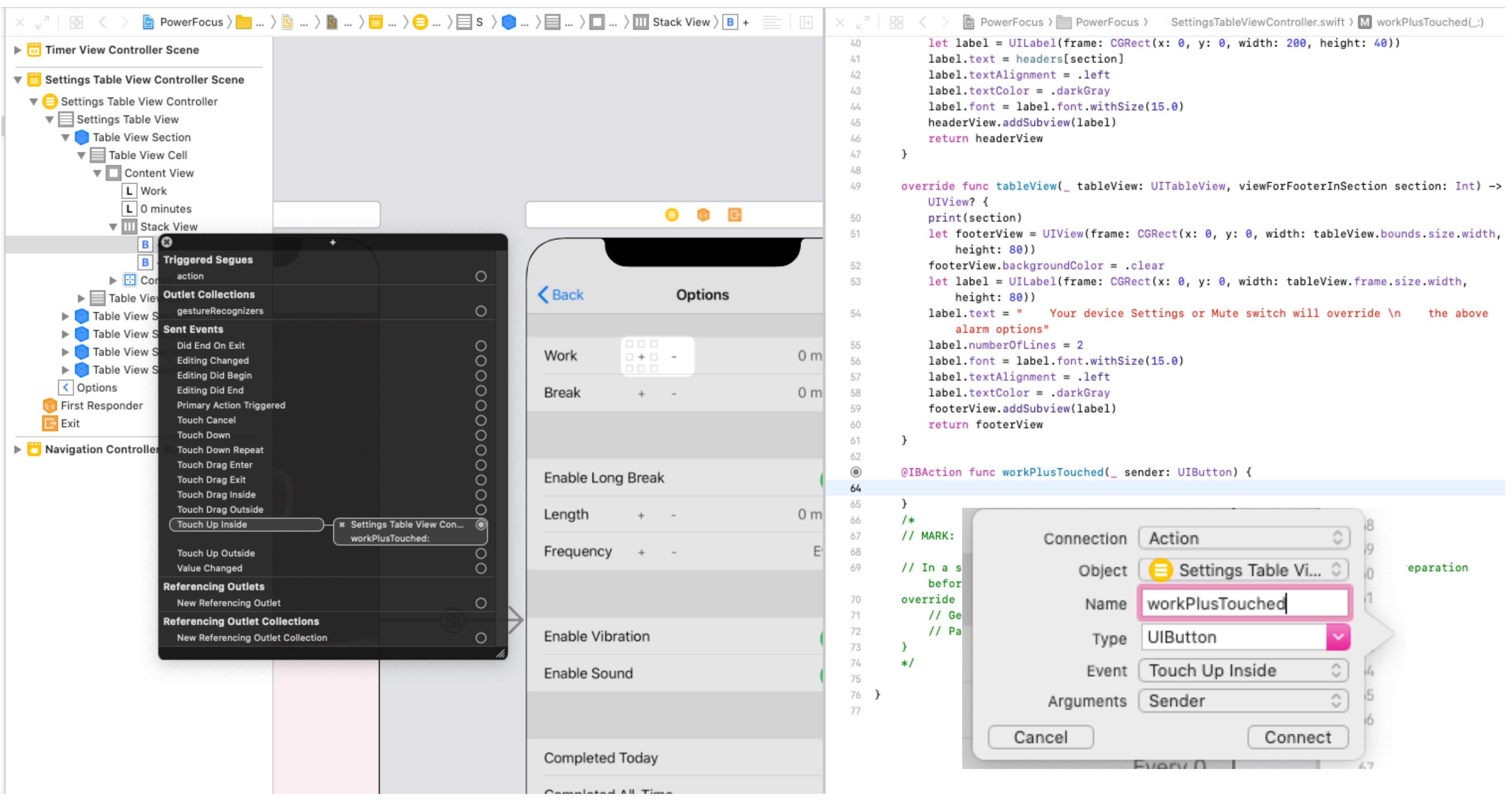
III. MODIFY USER DEFAULTS IN SETTINGS

The SettingsViewController also need to access the user defaults. Just like what we did in the View Controller, add an attribute for standard user default like this:

```
private var userDefaults: UserDefaults = .standard
```

Additionally, the SettingsViewController also need to set the userDefault when the user toggles the enable switches or touches the "+" or "-" buttons. To do so, we need to create actions and outlets of these widgets in the SettingsViewController.

Control-drag from the button to the SettingsViewController to add an action of the work plus button.



```
        let label = UILabel(frame: CGRect(x: 0, y: 0, width: 200, height: 40))
        label.text = headers[section]
        label.textAlignment = .left
        label.textColor = .darkGray
        label.font = label.font.withSize(15.0)
        headerView.addSubview(label)
        return headerView
    }

    override func tableView(_ tableView: UITableView, viewForFooterInSection section: Int) ->
        UIView? {
        print(section)
        let footerView = UIView(frame: CGRect(x: 0, y: 0, width: tableView.bounds.size.width,
            height: 80))
        footerView.backgroundColor = .clear
        let label = UILabel(frame: CGRect(x: 0, y: 0, width: tableView.frame.size.width,
            height: 80))
        label.text = "    Your device Settings or Mute switch will override \n    the above
            alarm options"
        label.numberOfLines = 2
        label.font = label.font.withSize(15.0)
        label.textAlignment = .left
        label.textColor = .darkGray
        footerView.addSubview(label)
        return footerView
    }

    @IBAction func workPlusTouched(_ sender: UIButton) {

    }
```

Connection: Action
Object: Settings Table Vi...
Name: workPlusTouched
Type: UIButton
Event: Touch Up Inside
Arguments: Sender

Cancel | Connect

The method workPlusTouched increments the value of work length in user default. First we get the current value of "workLength", if it's less than 59, we increment its value by one. So the max value of workLength is 59.

no more than 59. In that method, update the user default value of work:

```
@IBAction func workPlusTouched(_ sender: UIButton) {
    let currentWorkLength = userDefaults.integer(forKey: "workLength")
    if currentWorkLength < 59 {
        userDefaults.set(currentWorkLength + 1, forKey: "workLength")
    }
}
```

Do the same thing for decrementing the work length, add an action for the minus button and add the following method:

```
@IBAction func workMinusTouched(_ sender: UIButton) {
    let currentWorkLength = userDefaults.integer(forKey: "workLength")
    if currentWorkLength > 5 {
        userDefaults.set(currentWorkLength - 1, forKey: "workLength")
    }
}
```

Add an outlet of the work UIlabel that indicates the work length to the settings view controller:

```
@IBOutlet weak var workLengthLabel: UILabel!
```

There's a label workLengthLabel in the UI to show the current value of work length. When the value is modified by the actions, the UI label should be updated. The label should also get initialized when the view is loaded. You should also update all other labels on data changes, so manually updating each field is quite inconvenient.

A common practice is to create a method that updates all the outlets from user defaults. It gets called when there's a change and when the view is loaded. Using the update method is way clearer, easier to synchronize, and prevent missing an update.

Add the following method to SettingsViewController:

```
private func updateOutlets() -> Void {
    workLengthLabel.text = "\(userDefaults.integer(forKey: "workLength")) minutes"
    // more to go
}
```

Call this method at the end of the viewDidLoad method:

```
override func viewDidLoad() {
    super.viewDidLoad()
    . . .
    updateOutlets()

}
```

Then we only have to call it when an action triggers an update. It is slightly less efficient because it updates all the outlets but it makes the code much clearer and much easier to maintain by avoiding inconsistent outlets.

```
@IBOutlet weak var workLengthLabel: UILabel!
@IBAction func workPlusTouched(_ sender: UIButton) {
    let currentWorkLength = userDefaults.integer(forKey: "workLength")
    if currentWorkLength < 59 {
        userDefaults.set(currentWorkLength + 1, forKey: "workLength")
        updateOutlets()
    }
}

@IBAction func workMinusTouched(_ sender: UIButton) {
    let currentWorkLength = userDefaults.integer(forKey: "workLength")
```

```
        if currentWorkLength > 5 {
            userDefaults.set(currentWorkLength - 1, forKey: "workLength")
            updateOutlets()
        }
    }
```

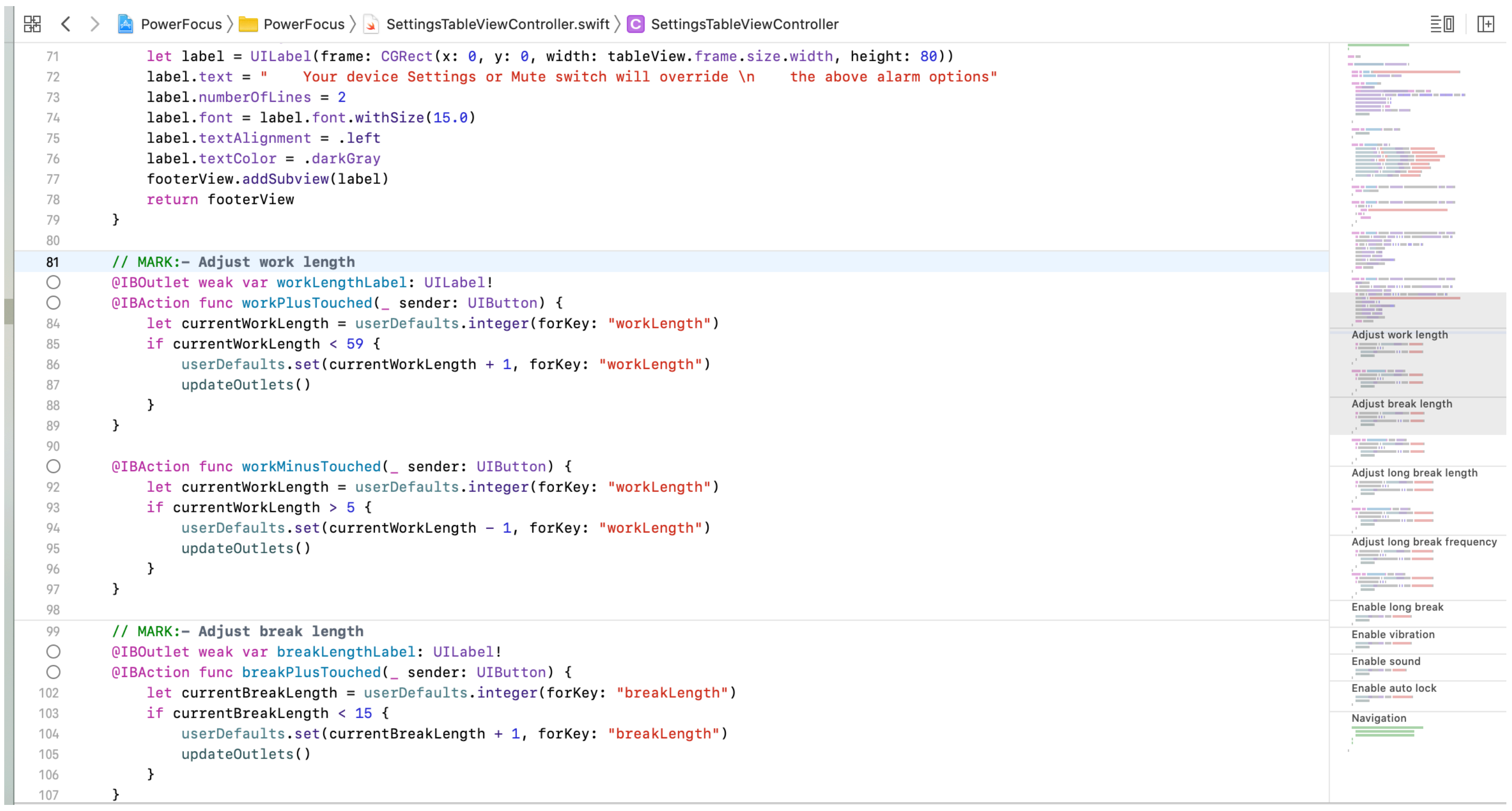
Do the same thing for break length, long break length, and long break frequency. Here are the main steps:

1. Create the outlet and action method between the Settings view and the SettingsViewController.
2. Update the user default values in the action method, pay attention to the intervals.
3. Update the outlet in the method updateOutlets.
4. Call updateOutlets in the action method.

The method updateOutlets should look like this with all the labels to update:

```
    private func updateOutlets() -> Void {
        workLengthLabel.text = "\(userDefaults.integer(forKey: "workLength")) minutes"
        breakLengthLabel.text = "\(userDefaults.integer(forKey: "breakLength")) minutes"
        longBreakLengthLabel.text = "\(userDefaults.integer(forKey: "longBreakLength")) minutes"
        frequencyLabel.text = "Every \(userDefaults.integer(forKey: "longBreakFrequency"))"
    }
```

When the file gets longer and longer, you can use the special comment "Mark" to make it easier to navigate in the file. You should put the groups of similar functionality in the same place and add a mark in front of that group. You can also preview the marks in the mini map on the right.



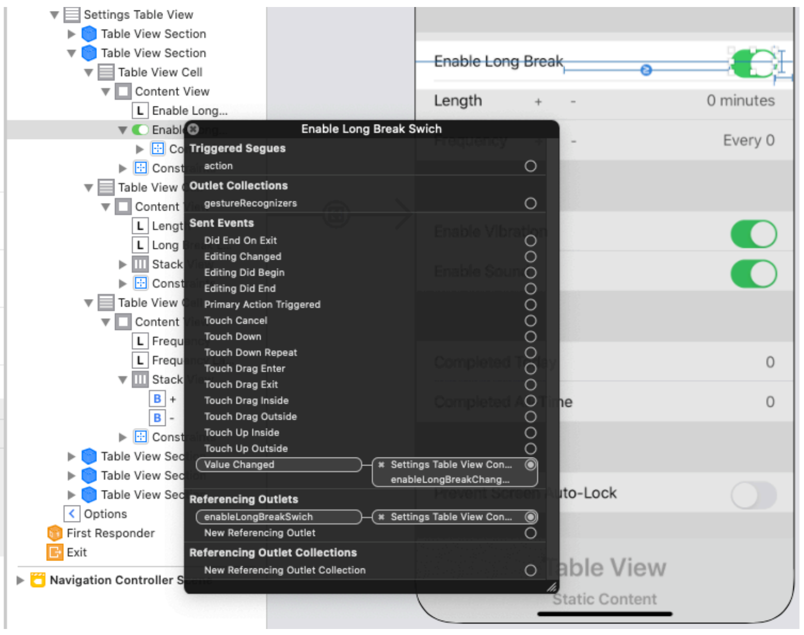
```
        let label = UILabel(frame: CGRect(x: 0, y: 0, width: tableView.frame.size.width, height: 80))
        label.text = "    Your device Settings or Mute switch will override \n    the above alarm options"
        label.numberOfLines = 2
        label.font = label.font.withSize(15.0)
        label.textAlignment = .left
        label.textColor = .darkGray
        footerView.addSubview(label)
        return footerView
    }

    // MARK:- Adjust work length
    @IBOutlet weak var workLengthLabel: UILabel!
    @IBAction func workPlusTouched(_ sender: UIButton) {
        let currentWorkLength = userDefaults.integer(forKey: "workLength")
        if currentWorkLength < 59 {
            userDefaults.set(currentWorkLength + 1, forKey: "workLength")
            updateOutlets()
        }
    }

    @IBAction func workMinusTouched(_ sender: UIButton) {
        let currentWorkLength = userDefaults.integer(forKey: "workLength")
        if currentWorkLength > 5 {
            userDefaults.set(currentWorkLength - 1, forKey: "workLength")
            updateOutlets()
        }
    }

    // MARK:- Adjust break length
    @IBOutlet weak var breakLengthLabel: UILabel!
    @IBAction func breakPlusTouched(_ sender: UIButton) {
        let currentBreakLength = userDefaults.integer(forKey: "breakLength")
        if currentBreakLength < 15 {
            userDefaults.set(currentBreakLength + 1, forKey: "breakLength")
            updateOutlets()
        }
    }
```

Similar to buttons, create an outlet and an action for the enable long break switch. We need the action to handle the value change and we need the outlet to initialize according to the initial state.



Enable Long Break Swich

Triggered Segues
action

Outlet Collections
gestureRecognizers

Sent Events
Did End On Exit
Editing Changed
Editing Did Begin
Editing Did End
Primary Action Triggered
Touch Cancel
Touch Down
Touch Down Repeat
Touch Drag Enter
Touch Drag Exit
Touch Drag Inside
Touch Drag Outside
Touch Up Inside
Touch Up Outside
Value Changed — Settings Table View Con... enableLongBreakChang...

Referencing Outlets
enableLongBreakSwich — Settings Table View Con...
New Referencing Outlet

Referencing Outlet Collections
New Referencing Outlet Collection

When a switch is toggled, the action “value changed” is triggered. The attribute UISwitch.isOn: bool indicates if the switch is enabled. Add the following code to setting view controller:

```
    // MARK: - Enable long break
    @IBOutlet weak var enableLongBreakSwich: UISwitch!
    @IBAction func enableLongBreakChanged(_ sender: UISwitch) {
        userDefaults.set(sender.isOn, forKey: "enableLongBreak")
        updateOutlets()
    }
```

In the method updateOutlets, update the attribute isOn according to the boolean value in user default:

```
    private func updateOutlets() -> Void {
        . . .
        enableLongBreakSwich.isOn = userDefaults.bool(forKey: "enableLongBreak")
    }
```

Here we set the attribute UISwitch.isOn to the boolean value in userDefaults. If the value in userDefaults is different from the isOn state of the UISwitch component, they are then out of sync. The synchronization won’t stuck in a loop (the method enableLongBreakChanged and updateOutlets call each other over and over again) because the method enableLongBreakChanged is only triggered when the UISwitch is toggled via user interface, not via code.

Do the same thing for enableSound, enableVibration, and preventScreenLock. The method updateOutlets should become:

```
    private func updateOutlets() -> Void {
        workLengthLabel.text = "\(userDefaults.integer(forKey: "workLength")) minutes"
        breakLengthLabel.text = "\(userDefaults.integer(forKey: "breakLength")) minutes"
        longBreakLengthLabel.text = "\(userDefaults.integer(forKey: "longBreakLength")) minutes"
        frequencyLabel.text = "Every \(userDefaults.integer(forKey: "longBreakFrequency"))"
        enableLongBreakSwich.isOn = userDefaults.bool(forKey: "enableLongBreak")
        enableVibrationSwitch.isOn = userDefaults.bool(forKey: "enableVibration")
        enableSoundSwitch.isOn = userDefaults.bool(forKey: "enableSound")
        lockSwitch.isOn = userDefaults.bool(forKey: "enablePreventLock")
    }
```

All the outlets should be updated when the view is loaded for the first time and when the view is about to appear each time. For now we only call the updateOutlets method in viewDidLoad method, which only gets called once when the settings table view is created and loaded. But the user can go between the main view and the settings table view several times, so we need to update the settings every time the settings table view is about to appear. Among the view controller lifecycle methods, we need to override the viewWillAppear method to update the outlets.

Add the following override method to the class SettingsViewController:

```
    override func viewWillAppear(_ animated: Bool) {
        updateOutlets()
    }
```

Now you can test it out. Click options button and change some settings in the table view. If you have done everything correctly, your changes will persist and update correctly when you touch the buttons or toggle the switches.

IV. UPDATE TIMER IN VIEW CONTROLLER

In the main view controller, we need to take into consideration that the focusTimer has just been updated when the view is about to appear. There are two cases to consider:

- Come back to a stopped state: The user went to options when the timer is stopped. We need to initialize the focusTimer according to the user defaults values and update the time label because the user might change the work length in options.
- Come back to a paused state: The timer was in pause or got paused when the user left the main view. We only need to update the focusTimer.

So in the viewWillAppear method, add the following lines of code:

```
    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)
        self.navigationController?.setNavigationBarHidden(true, animated: true)
        if timerStatus == .stopped {
            initFocusTimer()
            timeLabel.text = "\(userDefaults.integer(forKey: "workLength")):00"
        } else {
            updateFocusTimer()
        }
    }
```

In the method initFocusTimer, the focusTimer objects is created with the up-to-date values. Each time the view is loaded, this should get called.

```
private func initFocusTimer() -> Void {
    focusTimer = FocusTimer(3, 1, 2, 3, true)
    if userDefaults.object(forKey: "workLength") == nil {
        // Set default timer attributes for the first launch
       userDefaults.set(. . .)
    } else {
        focusTimer = FocusTimer(userDefaults.integer(forKey: "workLength"),
            userDefaults.integer(forKey: "breakLength"),
            userDefaults.integer(forKey: "longBreakLength"),
            userDefaults.integer(forKey: "longBreakFrequency"),
            userDefaults.bool(forKey: "enableLongBreak"))
    }

}
```

In the method updateFocusTimer, we don't actually create a new focusTimer object because we need the memorized attributes like the time left, the cycle count… The values like the lengths and frequencies could change before the view appears, so we simply use the setters of the focusTimer to set the up-to-date values in userDefaults.

```
func updateFocusTimer() -> Void {
    focusTimer.setWorkLength(to: userDefaults.integer(forKey: "workLength"))
    focusTimer.setBreakLength(to: userDefaults.integer(forKey: "breakLength"))
    focusTimer.setLongBreakLength(to: userDefaults.integer(forKey: "longBreakLength"))
    focusTimer.setLongBreakFreq(to: userDefaults.integer(forKey: "longBreakFrequency"))
    focusTimer.setLongBreakEnabled(to: userDefaults.bool(forKey: "enableLongBreak"))

}
```

Now launch the application, see what happens when you change the values in settings from a stopped timer and a paused timer. The stopped timer should give you the new work length value and the paused timer should keep the remaining time. When you resume or start the timer, the timer should behave according to the updated values.

EXERCISE #1

In the two classes that use user defaults, we used the strings as the key to access or update a user default entry. However, if we mistyped a string somewhere, it will return the default value of the getter type (0 for int and double, “” for string, false for boolean). This can lead to potential bugs.

A common practice is to use an enum to store used user default keys. Create a new Swift file in the PowerFocus group and name it DefaultKeys.swift. Create an enum called DefaultKeys and make it adopt String protocol. This means you have to assign each case a string value, and there isn’t other possible cases except the listed ones.

```
enum DefaultKeys: String {
    case workLength = "workLength"
    // ...
}
```

In the view controllers, whenever you want to access the userDefaults by key, you should use one of the cases of DefaultKeys. So instead of calling:

```
userDefaults.set(25, forKey: “workLength")
```

You can use:

```
userDefaults.set(25, forKey: DefaultKeys.workLength.rawValue)
```

Please complete the DefaultKeys enumeration and use its cases correctly in ViewController and SettingsTableViewController.

EXERCISE #2

Copy and paste your code are usually considered a bad practice for software engineers, but sometimes it's just inevitable. When there are some pieces of code that share much similarity, you should always ask yourself this question: Can I avoid this by reorganizing my code or changing my code structure? If the answer is yes, you should do it in a heartbeat because writing and maintaining code with large similar chunks is very unpleasant and can lead to some serious bugs in the future.

We have created outlets for the plus button, the minus button, and the label for the work length. We also have created two outlets to handle the button clicks. In this exercise, do the same thing for the break length, the long break length, and the long break frequency. Also create an outlet and an action for the enable vibration, enable sound, and prevent screen auto-lock switch.

You will soon realize that it becomes quite tedious and repetitive. The pieces of code are almost the same and you have an urge to use copy and paste. You can do that and change some values manually like the +/- sign of increment/decrement methods, the keys for user defaults…

In chapter eleven, we have learnt how to write unit tests. Write some unit tests to check if you have missed a spot to change. For instance, if you leave in breakPlusTouched method the key of DefaultKeys.workLength, the corresponding updated attribute will be incorrect.

So the lesson is, you can use copy and paste, but you should write some unit tests to check if you have missed a spot to change.

SUMMARY

In this chapter, we have worked on these topics:

- Ways to achieve data persistence
- Persistence implementation with user defaults
- Make settings responsive
- Update timer and user defaults

If any of these is unclear to you, please make sure to go back and read the related part or parts before moving on the the next chapter.

If you can't figure out the exercise on your own, you can check the answers at

https://github.com/CarlistleZ/MyiOSTutorial/blob/master/Chapter_Sixteen_Exercises.zip

For reference, you can download the complete Xcode project from

https://github.com/CarlistleZ/MyiOSTutorial/blob/master/Chapter_Sixteen_PowerFocus.zip