CHAPTER FOURTEEN : INTERACTION MANAGEMENT AND MODEL PERSISTENCE

I. INTRODUCTION

In the previous chapter, we added some basic functionalities to the view controller and a timer of iOS foundation framework. The view controller is able to update the labels and buttons and react to the button touched events. But when we were trying to add some features like making the timer pause and resume, the updates got too complex and somehow buggy.

Situations like this are quite common in the view controller implementations. iOS developers normally classify interaction cases. Once these cases are sorted out, the can be developed and tested one by one. Being able to break down a huge view controller into several simpler interaction cases makes the view controller easier to develop, understand, and maintain.

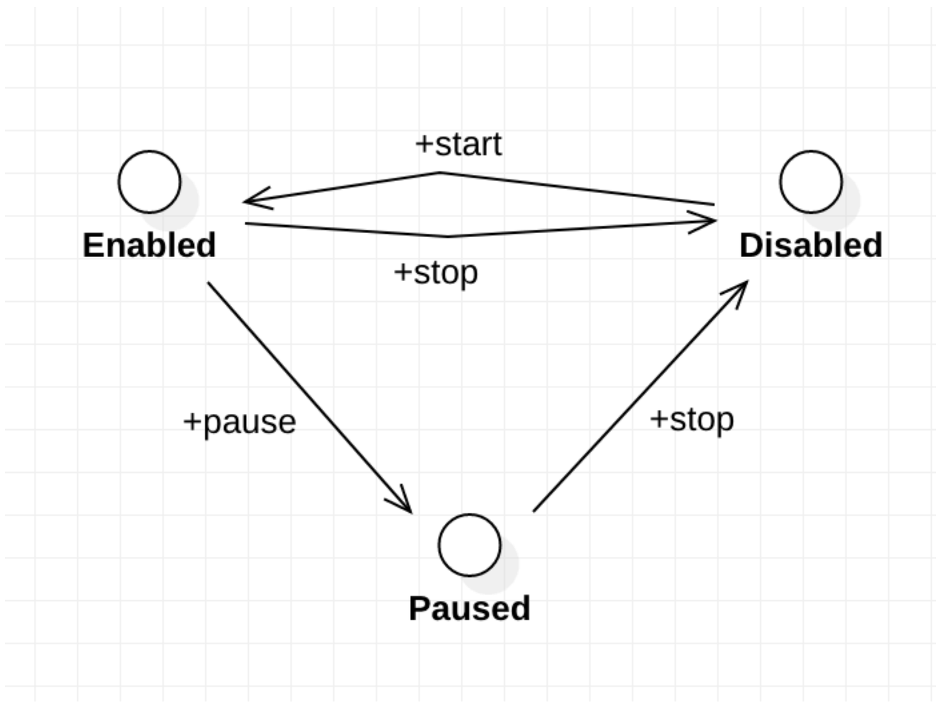
When the model is updated by the view controller and the view gets resigned, how do we keep the model changes? iOS provides Core Data to implement model persistence. Core Data is a framework that you use to manage the model layer objects in your application. It provides generalized and automated solutions to common tasks associated with object life cycle and object graph management, including persistence. We will use Core Data on FocusTimer in this chapter.

II. INTERACTION CASES

When the timer view is displayed, what are the possible states of the timer?

- The timer can be **disabled**. When the page is loaded, the timer is dibbled by default. When the stop button is clicked, the timer is also disabled. From this state, the timer can go to the start state.
- The timer can be **enabled**. In this state, the timer's handler is called to decrement the time left. When the start button is clicked from the disabled state or when the resume button is clicked from the paused state, the timer becomes enabled.
- The timer can be **paused**. When the timer is enabled, clicking the pause button

If we represent the states in a diagram of automata, it should look like this:

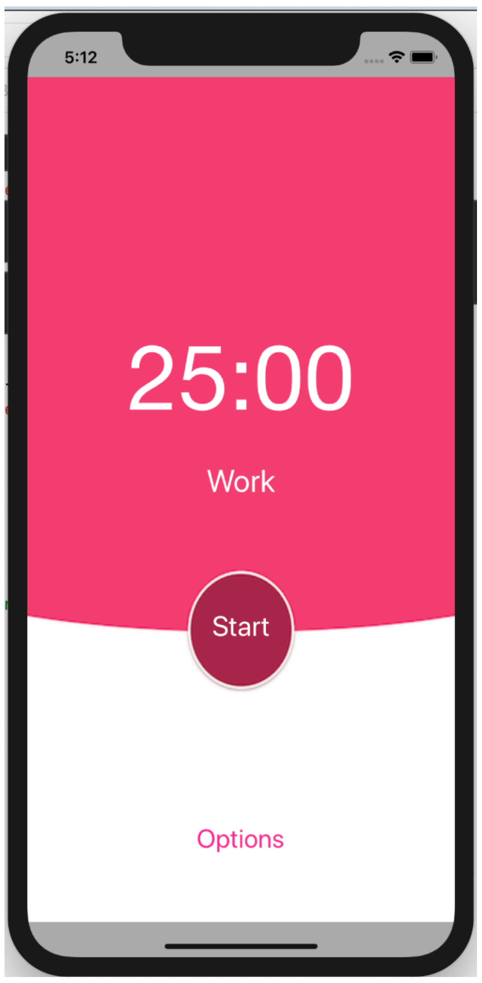


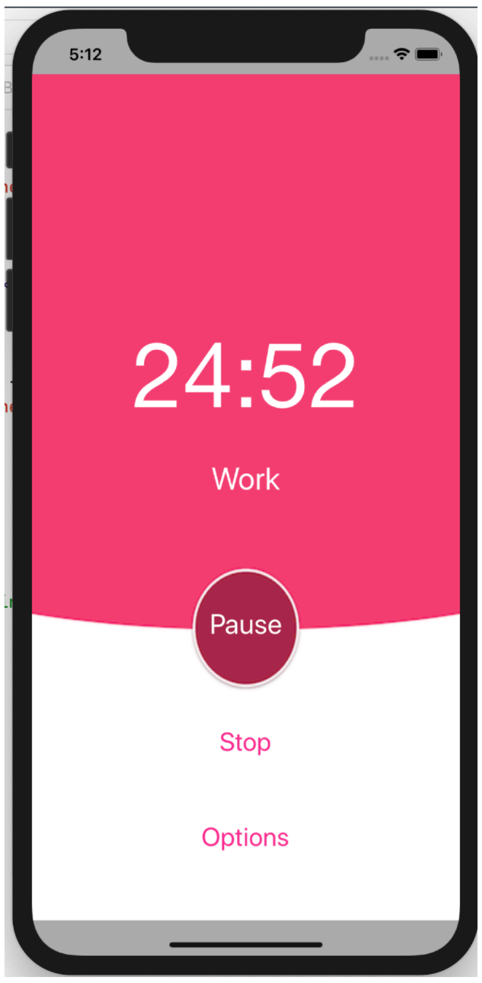
In the switch statement of the function start ButtonTouched, we should handle the transitions among the states.

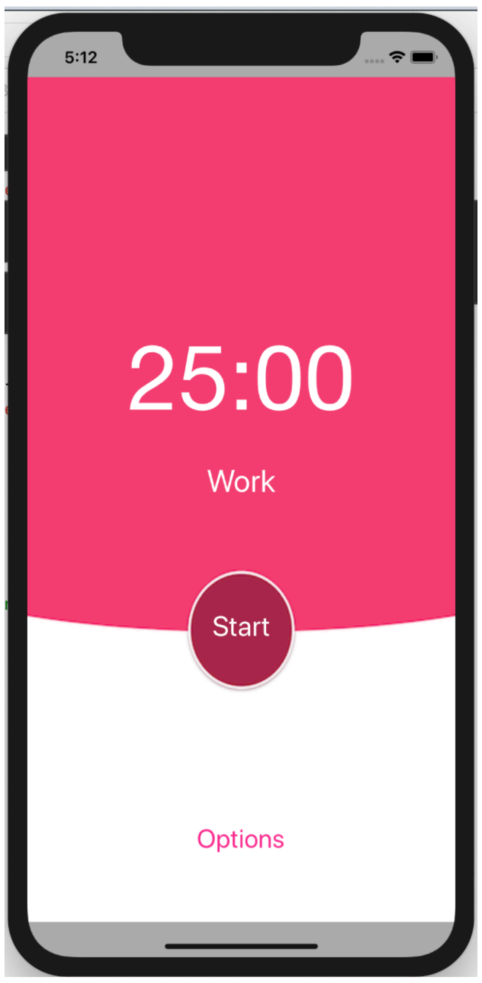
```
@IBAction func startButtonTouched(_ sender: UIButton) {
     switch timerStatus {
          …
     }
}
```

When the timer is stopped, clicking the start button will start the timer. To start the timer, we should set the title of the start button to pause (when the timer is enabled, the start button should become a pause button). The timerStatus should become .enabled. When the timer is active, stopButton should not be hidden. The timer should start from work, so the secondLeft attribute should be the length of a work activity.

```
     case .stoped:
```







```
        // start the timer from stoped
        startButton.setTitle("Pause", for: .normal)
        timerStatus = .enabled
        statusLabel.text = "Work"
        focusTimer.setSecondLeft(to: focusTimer.workLength * 60)
        stopButton.isHidden = false
```

When the timer is enabled, clicking the pause button will pause the timer. To pause the timer, we should set the timerStatus to .enabled. The start button becomes the resume button so its text should be “Resume”.

```
    case .enabled:
        // pause the timer from enabled
        startButton.setTitle("Resume", for: .normal)
        statusLabel.text = "Work paused"
        timerStatus = .paused
```

When the timer is paused, clicking the resume button will resume the timer. To resume the timer, we should set the timerStatus to .enabled. The start button becomes the pause button so its text should be “Pause”.

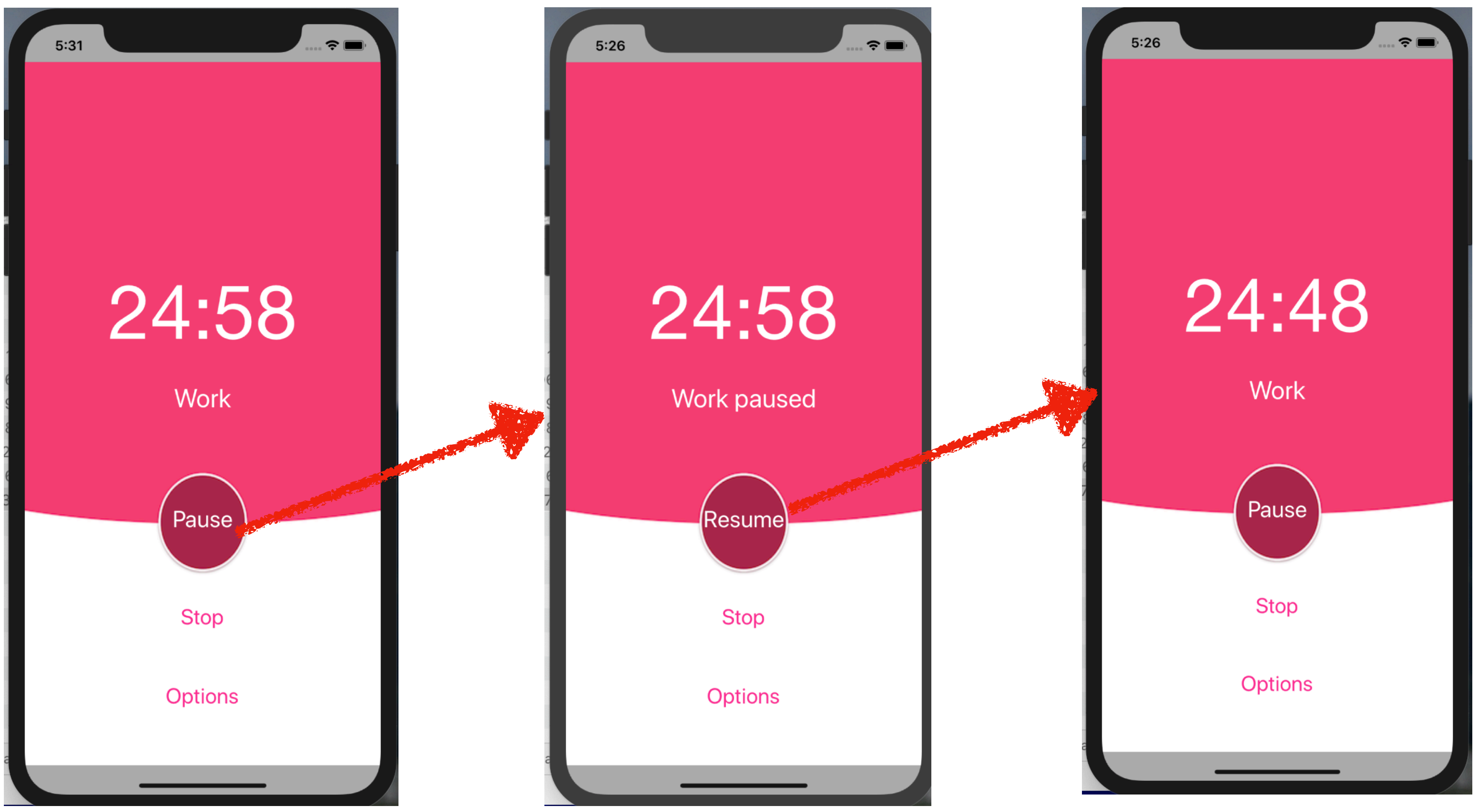
```
    case .paused:
        // resume the timer from paused
        startButton.setTitle("Pause", for: .normal)
        statusLabel.text = "Work"
        timerStatus = .enabled
```

In the method stopButtonTouched, the timer is set to stoped from the “enabled” or “paused” state. The time should become a full workLength, so the text of the timeLabel and the secondLeft attribute of the focusTimer should be set to focusTimer.workLength. When the timer is stopped, we can't click on it again so it should be hidden. And the timerStatus attribute of the viewController should be .stopped.

```
timeLabel.text = "\(focusTimer.workLength):00"
timerStatus = .stopped
statusLabel.text = "Work"
startButton.setTitle("Start", for: .normal)
focusTimer.setSecondLeft(to: focusTimer.workLength * 60)
stopButton.isHidden = true
```

Now let's test it out. From the initial state click start, the timer should become active. The countdown starts and the stop button appears. From the active state click stop, the timer should go back to the initial state. The stop button disappears.

From the initial state click start, the timer should become active. The countdown starts and the stop button appears. From the active state click pause, the timer should stop (not set to 25:00 but stop decrementing). The status label should change from "work" to "work

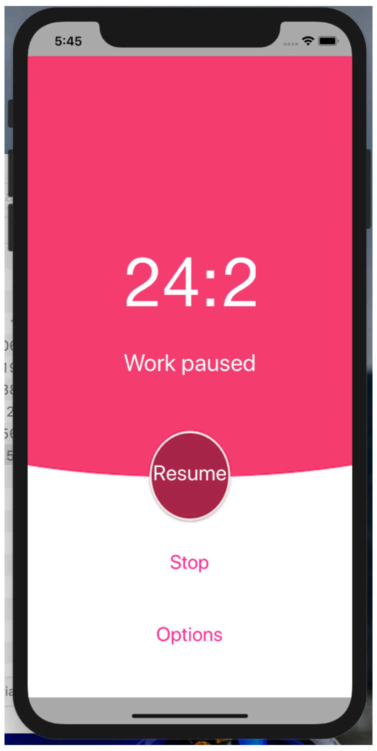


paused". Click resume, the status label should go back to "work" and the time label starts decrementing again.

III. USE INT EXTENSION TO FORMAT STRING

In chapter seven, we have seen an example of the extension mechanism. Extension is such a powerful tool in Swift, and here is a way of using extension to make the code much clearer and easier to understand.

In the current version, when the second goes to single digit, the label's second goes to single digit too. This looks weird because it goes against the convention that we're already used to. For instance, if the time is 24min02sec, we have the following result:



In the handler decSec, the time label is set like this:

```
let sec = focusTimer.getSecondLeft()
timeLabel.text = "\(sec / 60):\( sec % 60)"
```

Here we can add an extension to Int. In this extension, we need a method that formats the int value to a string of "mm:ss". The extension syntax looks like this:

```
extension Int {
    func timeFormat() -> String {
        if self % 60 >= 10 {
            return "\(self / 60):\( self % 60)"
        } else {
            return "\(self / 60):0\( self % 60)"
        }
    }
}
```

The method timeFormat() -> String is available for all integers, so you can use it like 5.timeFormat() and you will get a string of “00:05”. In the method timeFormat, to access the integer value of the object (which is itself), use the keyword “self”. So if the remainder (second) is less than 10, we add an extra “0” before the remainder. So the string is always in the format of “mm:ss”.

In the handler, instead of formatting the sec directly, you can simply call the extended method because the variable sec is an instance of Int.

```
timeLabel.text = sec.timeFormat()
```

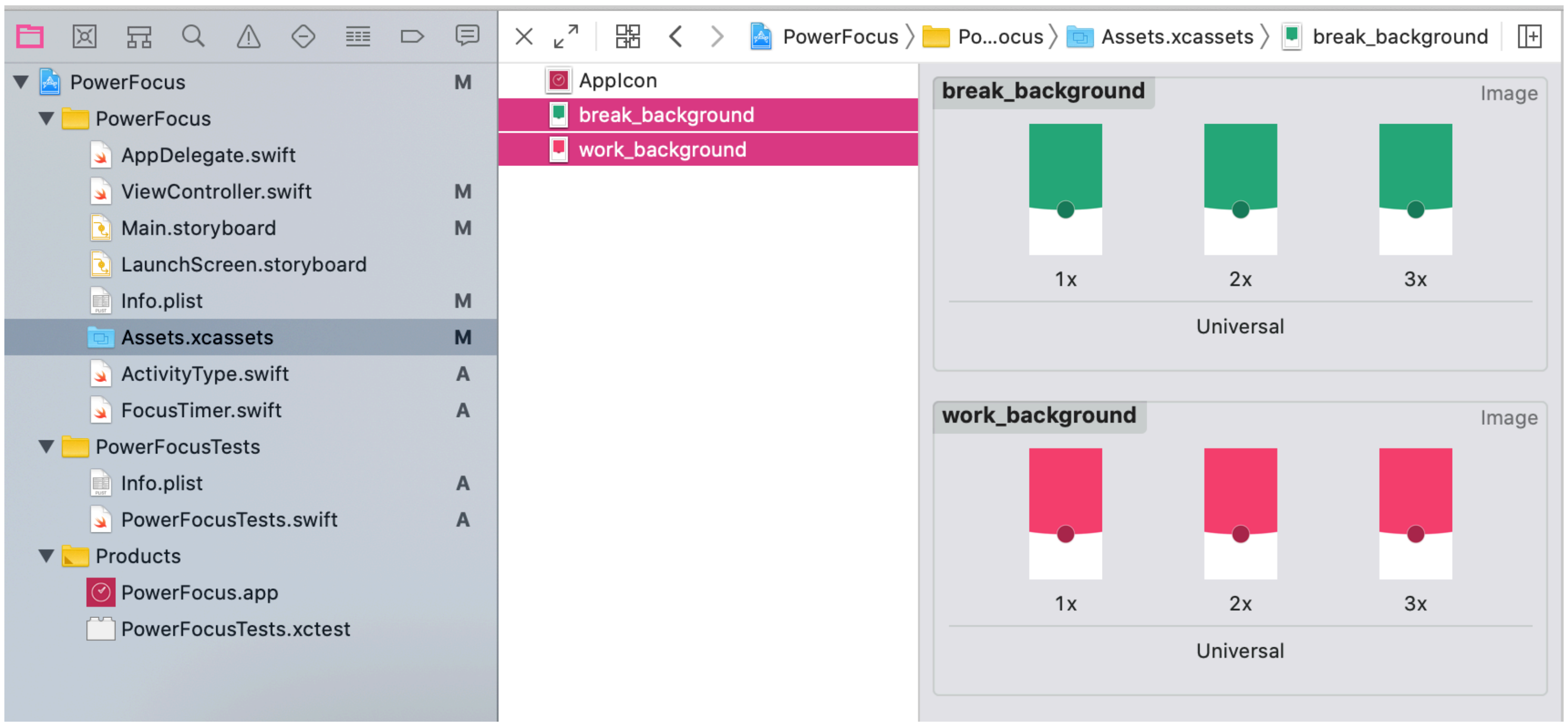
IV. ALTERNATE BETWEEN WORK AND PAUSE

When the time is up for an activity, the timer should change the theme to the following activity. The theme is red for the work activity, and green for the break activity.

The first thing to do is to add the background image assets. Click the following link to download the assets and add them to your Assets.xcassets folder.

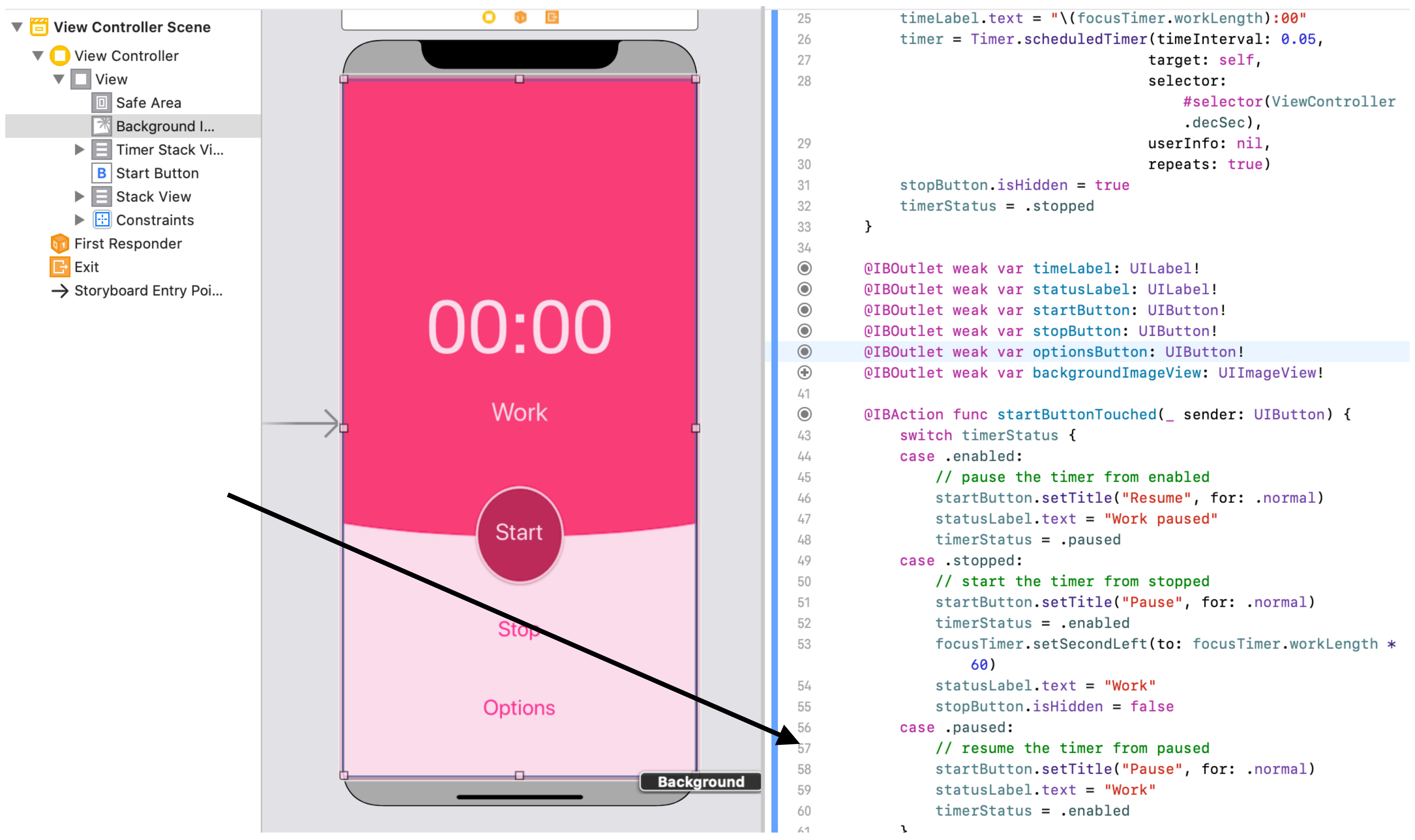
https://github.com/CarlistleZ/MyiOSTutorial/blob/master/Chapter_Fourteen_Background_Asset.zip

The image asset folder should look like this:



In the View Controller, add a method “setTheme” to set the theme of the storyboard.

```
func setWorkTheme(to activityType: ActivityType) -> Void {
        switch activityType {
        case .work:
            <#code#>
        case .longBreak:
            <#code#>
        case .shortBreak:
            <#code#>
```



```
        timeLabel.text = "\(focusTimer.workLength):00"
        timer = Timer.scheduledTimer(timeInterval: 0.05,
                                     target: self,
                                     selector:
                                        #selector(ViewController
                                        .decSec),
                                     userInfo: nil,
                                     repeats: true)
        stopButton.isHidden = true
        timerStatus = .stopped
    }

    @IBOutlet weak var timeLabel: UILabel!
    @IBOutlet weak var statusLabel: UILabel!
    @IBOutlet weak var startButton: UIButton!
    @IBOutlet weak var stopButton: UIButton!
    @IBOutlet weak var optionsButton: UIButton!
    @IBOutlet weak var backgroundImageView: UIImageView!

    @IBAction func startButtonTouched(_ sender: UIButton) {
        switch timerStatus {
        case .enabled:
            // pause the timer from enabled
            startButton.setTitle("Resume", for: .normal)
            statusLabel.text = "Work paused"
            timerStatus = .paused
        case .stopped:
            // start the timer from stopped
            startButton.setTitle("Pause", for: .normal)
            timerStatus = .enabled
            focusTimer.setSecondLeft(to: focusTimer.workLength *
                60)
            statusLabel.text = "Work"
            stopButton.isHidden = false
        case .paused:
            // resume the timer from paused
            startButton.setTitle("Pause", for: .normal)
            statusLabel.text = "Work"
            timerStatus = .enabled
```

```
        }
    }
```

In the storyboard, we have an image view called BackgroundImageView. Create an outlet in the View Controller like this:

In the method func setWorkTheme(to activityType: ActivityType) -> Void , we can use the image view outlet like this:

```
backgroundImageView.image = UIImage(named: “work_background”) // to set to work
backgroundImageView.image = UIImage(named: "break_background") // to set to break
```

To make the color of the button title match the theme, set the color of the titles as well when updating the theme:

```
    func setTheme(to activityType: ActivityType) -> Void {
        let greenColor =  colorLiteral(red: 0.3215686275, green: 0.6470588235, blue: 0.4823529412,
alpha: 1)
        let redColor =  colorLiteral(red: 0.8941176471, green: 0.3098039216, blue: 0.4470588235,
alpha: 1)
        switch activityType {
        case .work:
            backgroundImageView.image = UIImage(named: "work_background")
            stopButton.titleLabel?.textColor = redColor
            optionsButton.titleLabel?.textColor = redColor
        case .longBreak:
            backgroundImageView.image = UIImage(named: "break_background")
            stopButton.titleLabel?.textColor = greenColor
            optionsButton.titleLabel?.textColor = greenColor
        case .shortBreak:
            backgroundImageView.image = UIImage(named: "break_background")
            stopButton.titleLabel?.textColor = greenColor
            optionsButton.titleLabel?.textColor = greenColor
        }
    }
```

In the method @objc func decSec() -> Void, we have to deal with the situation where the focusTimer's current activity's time is up. We can use a switch on focusTimer's activity like this:

```
    @objc func decSec() -> Void {
        guard timerStatus == .enabled else {
            return
        }
        if focusTimer.decrementSecond() {
            switch focusTimer.getActivity() {
            case .longBreak:
                ..
            case .shortBreak:
                ..
            case .work:
                ..
            }
            let sec = focusTimer.getSecondLeft()
            timeLabel.text = sec.timeFormat()
        } else {
            let sec = focusTimer.getSecondLeft()
            timeLabel.text = sec.timeFormat()
        }
    }
```

For long breaks and short breaks, the completed cycles of the focus timer should be incremented. The focusTimer's activity should be work. We also need to set the theme and the status label to work. Lastly, the secondLeft of focusTimer should be a full work length (times 60 in seconds).

For work activity, there's a little bit more work to do. Suppose we are at the end of a work activity, how can we know whether the next break is a long break or a short break? An easy way to tell is compare the completeCycles to longBreakFreq.

Go to FocusTimer.swift and add the following method:

```
    func nextBreakIsLongBreak() -> Bool {
        return (completedCircles + 1) % longBreakFreq == 0
    }
```

The completedCircles + 1 represents the next cycle (when the work activity is over). The modulus operator checks if the completedCircles + 1 is a multiple of longBreakFreq. For example, for the long break frequency is 3, long breaks should happen in 3, 6, 9… So the completeCycles 2,5,8,… will give a positive result. So in the case .work, we can set the secondLeft attribute thanks to the method focusTimer.nextBreakIsLongBreak.

The complete switch looks like this:

```
switch focusTimer.getActivity() {
            case .longBreak:
                _ = focusTimer.incrementCompletedCycles()
                focusTimer.setActivity(to: .work)
                focusTimer.setSecondLeft(to: focusTimer.workLength * 60)
                statusLabel.text = "Work"
                setTheme(to: .work)
            case .shortBreak:
                _ = focusTimer.incrementCompletedCycles()
                focusTimer.setActivity(to: .work)
                focusTimer.setSecondLeft(to: focusTimer.workLength * 60)
                statusLabel.text = "Work"
                setTheme(to: .work)
            case .work:
                if focusTimer.nextBreakIsLongBreak() {
                    focusTimer.setActivity(to: .longBreak)
                    focusTimer.setSecondLeft(to: focusTimer.longBreakLength * 60)
                    setTheme(to: .longBreak)
                } else {
                    focusTimer.setActivity(to: .shortBreak)
                    focusTimer.setSecondLeft(to: focusTimer.breakLength * 60)
                    setTheme(to: .shortBreak)
                }
                statusLabel.text = "Break"
            }
```

If you add print statement before every change of event, you will get the following output (long break frequency is 3):

```
Start of: work
Start of: short break
Start of: work
Start of: short break
Start of: work
Start of: long break
Start of: work
Start of: short break
…
```

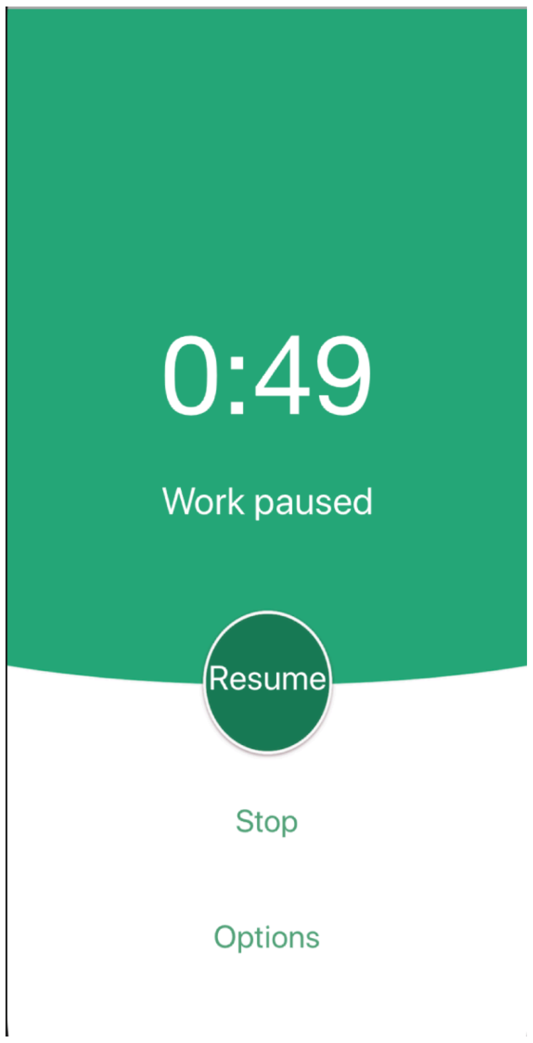
If it is not your case, go back to the handler and check each case. The background and the buttons should stay updated and have matching colors. If it's not the case, go back to the method setTheme and check it out.

Keeping everything in sync can be a hard thing during the development and checking the correctness can be quite tedious. When there's a bug, we should be able to locate its source and fix it quite easily if we've understand the code.

Here are two bugs caused by inconsistent state. Try to locate the source and fix them!

EXERCISE #1

In a break activity, click pause button, you will see that the state label says "Work paused". This is obviously incorrect. Now think about which part of the code can cause this problem. Go there and fix it.

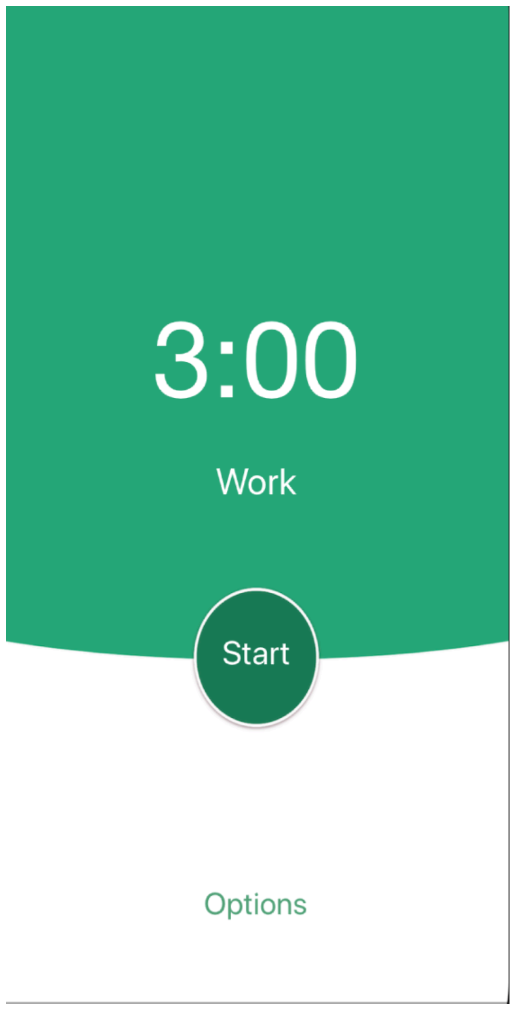


Hint: this is caused by the touch of a button, so you should look in buttons' @IBAction methods.

EXERCISE #2

In a break activity, click stop button, you will see that the theme stays green. We have an initialization problem. When the stop button is clicked, everything should be reinitialized.

Now think about which part of the code can cause this problem. Go there and fix it.



Hint: this is caused by an initialization issue, so you should look in the places where components are initialized.

SUMMARY

In this chapter, we have worked on these topics:

- specify timer's user interaction cases
- review usage of extension
- alternate UI themes
- practice debugging

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 exercises on your own, you can check the answers at

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

For reference, you can download the complete Xcode project from

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