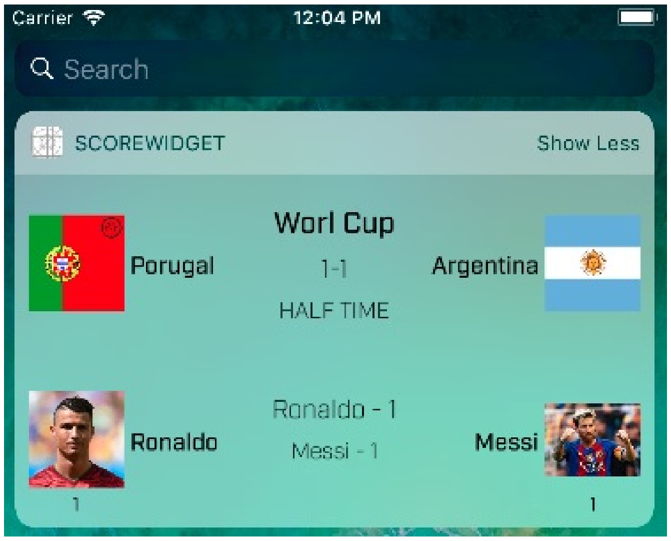
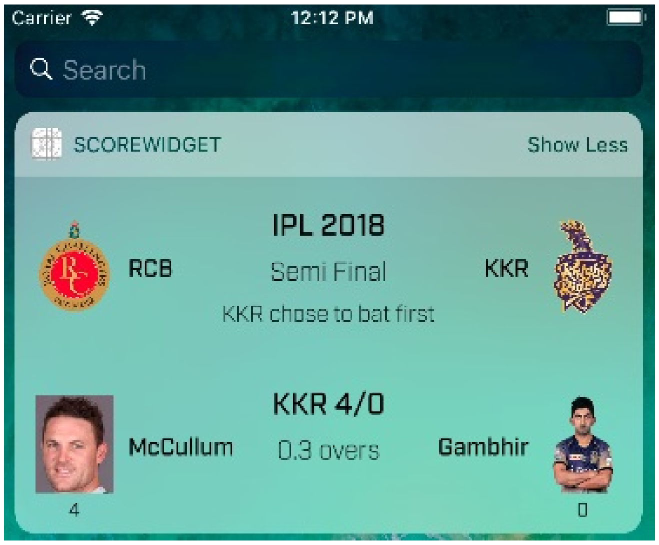
Today Extension: Score Widget



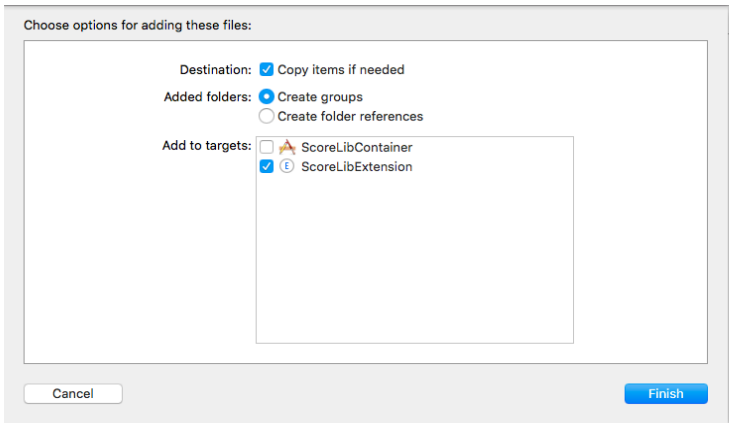


Score widget is a reusable component that can be used to display the scores and other details of any game through Today Extension. Extending your container app with Today Extension lets you engage user even when their phone is locked. Lets see how to use this cool widget.

Import Score Widget files in your project

After adding the new target (viz. Today Extension), you need to import the Score Widget files to your project. Simply unzip the ScoreWidget.zip, and drop the unzipped folder under your target's hierarchy.

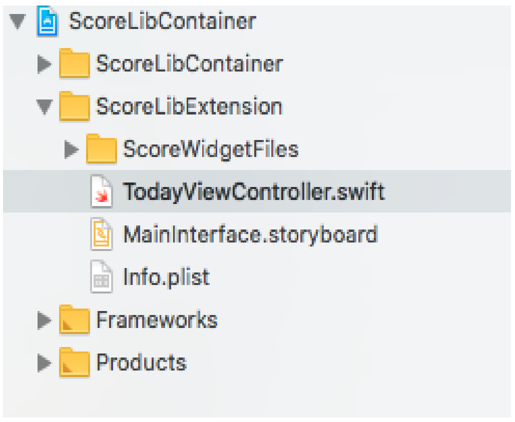
Note: Make sure that you select "Copy if needed", and extension as the target while adding the files.


Choose options for adding these files:

Destination: ✓ Copy items if needed

Added folders: ● Create groups
○ Create folder references

Add to targets: ☐ ScoreLibContainer
✓ ScoreLibExtension


```
ScoreLibContainer
  ScoreLibContainer
  ScoreLibExtension
    ScoreWidgetFiles
    TodayViewController.swift
    MainInterface.storyboard
    Info.plist
  Frameworks
  Products
```

Adding the Score Widget View

Once you add the Today Extension target to your project, XCode automatically generates *TodayViewController.swift* and *MainInterface.storyboard.*

Go to the *MainInterface.storyboard* and delete the Label from TodayViewControllerScene. We did not need this label as we are having our XIB for score widget view.

Next we need to add our XIB as subview to TodayViewControllerScene. To do so, add below as the first line of your TodayViewController class.

```
var scoreWidgetView:ScoreWidgetView?
```

```
class TodayViewController: UIViewController, NCWidgetProviding {
    var scoreWidgetView:ScoreWidgetView?
```

Instantiate the declared variable and add as subview using below code.

```
scoreWidgetView = ScoreWidgetView().loadNib() as? ScoreWidgetView
scoreWidgetView?.frame = self.view.frame
self.view.addSubview(scoreWidgetView!)
```

This is how your viewDidLoad() should look like.

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

        scoreWidgetView = ScoreWidgetView().loadNib() as?
ScoreWidgetView
        scoreWidgetView?.frame = self.view.frame
        self.view.addSubview(scoreWidgetView!)
    }
```

We are done with instantiating our custom view and adding it as subview. Now lets handle the expansion and shrinking of view on "Show More / Show Less" button.

Handling Expansion and Shrinking of View

To add the "Show More/ Show Less" button, add below lines to your viewDidLoad.

if #available(iOSApplicationExtension 10.0, *)
{
 extensionContext?.widgetLargestAvailableDisplayMode = .expanded
} else {
// Fallback on earlier versions
}

Value of 'widgetLargestAvailableDisplayMode' decides that whether the widget can be expanded or not. By now, your viewDidLoad() should look like

```
override func viewDidLoad() {
        super.viewDidLoad()
        if #available(iOSApplicationExtension 10.0, *) {
            extensionContext?.widgetLargestAvailableDisplayMode =
.expanded
        } else {
            // Fallback on earlier versions
        }
        scoreWidgetView = ScoreWidgetView().loadNib() as?
ScoreWidgetView
        scoreWidgetView?.frame = self.view.frame
        self.view.addSubview(scoreWidgetView!)
  }
```

Adding the above lines will enable "Show More" button on our widget. Next we need to handle the expansion and shrinking when user taps on "Show More" button.

To do so, we will implement a delegate method i.e. *widgetActiveDisplayModeDidChange(_ activeDisplayMode: NCWidgetDisplayMode, withMaximumSize maxSize: CGSize).* Add the below method.

```
@available(iOSApplicationExtension 10.0, *)
    func widgetActiveDisplayModeDidChange(_ activeDisplayMode:
NCWidgetDisplayMode, withMaximumSize maxSize: CGSize) {
        self.preferredContentSize = (activeDisplayMode == .expanded) ?
CGSize(width: self.view.frame.size.width, height: EXPANDED_HEIGHT) :
CGSize(width: self.view.frame.size.width, height: SHRINKED_HEIGHT)
        self.scoreWidgetView?.shouldDetailInfoViewBeHidden(state:
(activeDisplayMode == .expanded) ? false : true)
    }
```

Here we have assigned a new frame every time the value of activeDisplayMode is changed. We also make our Detail Info View hidden when user taps on show less by below line.

```
self.scoreWidgetView?.shouldDetailInfoViewBeHidden(state:
(activeDisplayMode == .expanded) ? false : true)
```

Populating Data in the Widget

We are done with most of the part. Now the only thing remaining is populating the data in our score widget. To do so, you need to have an object of class *GameDataModel.* Instantiate an object of GameDataModel with the data which you want to view in the widget.

To keep the things simple, here I am taking an array having two objects of *GameDataModel.* You can use any data structure to populate the data depending upon your requirements.

```
     var dummyCricketData = [GameDataModel]()


     dummyCricketData.append(GameDataModel(leftImageUrl:
"https://upload.wikimedia.org/wikipedia/en/thumb/4/42/Royal_Challengers_
Bangalore_Logo.svg/878px-Royal_Challengers_Bangalore_Logo.svg.png",
rightImageUrl:
"https://www.thesouledstore.com/public/theSoul/uploads/catalog/product/2
0170331171533-1.jpg", leftTitleText: "RCB", rightTitleText: "KKR",
leftSubtitleText: "", rightSubtitleText: "", headerText: "IPL 2018",
detailText: "Semi Final", moreDetailText: "KKR chose to bat first"))


     dummyCricketData.append(GameDataModel(leftImageUrl:
"http://p.imgci.com/db/PICTURES/CMS/203000/203043.1.jpg", rightImageUrl:
"http://iplstatic.s3.amazonaws.com/players/210/84.png", leftTitleText:
"McCullum", rightTitleText: "Gambhir", leftSubtitleText: "4",
rightSubtitleText: "0", headerText: "KKR 4/0", detailText: "0.3 overs",
moreDetailText: ""))
```

Now that we have populated data in our array, lets put it on our view. To do so, we have method viz. *populateViewWith(data:GameDataModel,viewType:ViewType)*

You need to pass the object of GameDataModel as first parameter and view type as second paramenter.

View Type is an enum having two cases i.e. BasicInfo and DetailInfo. They are nothing more than upper view of widget and lower view of widget respectively.

Lets call this method for our scoreWidgetView and pass the data we just populated.

```
scoreWidgetView?.populateViewWith(data: dummyCricketData[0], viewType:
.basicInfo)

scoreWidgetView?.populateViewWith(data: dummyCricketData[1], viewType:
.detailInfo)
```

That was all needed. Run the project and you should see you widget working fine.

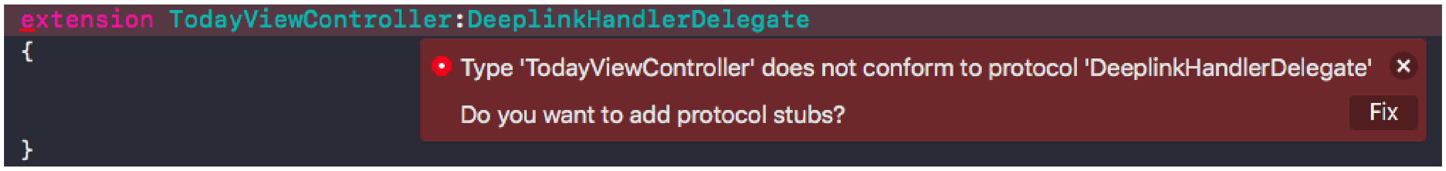
How about deep linking the widget to the container app ?

Opening container app on tap on widget

Once user taps on widget, we want him to navigate to the container app and land on screen containing details of the game tapped. This would be achieved by deep linking.

To know about the tap of user on widget, Score Widget provides some delegate methods. All you need to do is to confirm *DeeplinkHandlerDelegate* protocol.

To do so, write an extension of TodayViewController, implementing *DeeplinkHandlerDelegate* and say yes to XCode when it prompts the message for adding protocol stubs.



```
extension TodayViewController:DeeplinkHandlerDelegate
{

}
```

Type 'TodayViewController' does not conform to protocol 'DeeplinkHandlerDelegate'

Do you want to add protocol stubs?

Upon fixing this, XCode would write the delegate methods for you. Now add the below lines to the implementation of delegate methods.

let url = URL(string: "some deeplink url for basic info screen")
self.extensionContext?.open(url!, completionHandler: nil)

Your extension would look like.

```
      extension TodayViewController:DeeplinkHandlerDelegate
      {
          func navigateToBasicInfoScreen()
          {
              let url = URL(string: "some deeplink url for basic info
screen")
              self.extensionContext?.open(url!, completionHandler: nil)
          }
          func navigateToDetailInfoScreen()
          {
              let url = URL(string: "some deeplink url for detail info
screen")
              self.extensionContext?.open(url!, completionHandler: nil)
          }
      }
```

Note: Replace the string `"some deeplink url for basic info screen"` with your deeplink URL.

The last thing remaining is to set the delegate as self. Go to the viewDidLoad() and add below line

scoreWidgetView?.delegate = self

after the line in which scoreWidgetView has been instantiated.

```
scoreWidgetView = ScoreWidgetView().loadNib() as? ScoreWidgetView
        scoreWidgetView?.delegate = self
        scoreWidgetView?.frame = self.view.frame
        self.view.addSubview(scoreWidgetView!)
```

Run the project and click on the widget. If the deeplink URL you entered is correct, you should be navigated to the container app.

That's it. :)

Happy Coding !!!