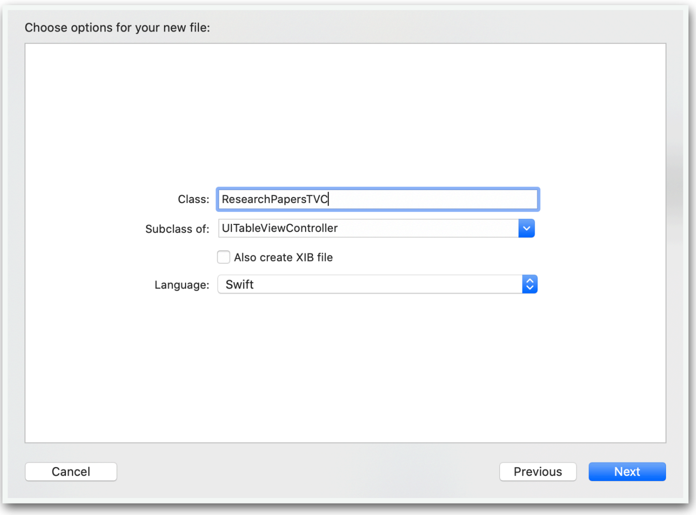
Week 9, w/c 20th November 2023

Research Papers – An API-based application

Create a new Xcode app swift project using Storyboard. Make sure you select the “Create Git repository on my Mac” option. Call the project “Research Papers”.

Open the storyboard. From the library locate a Table View Controller, and drag it to the storyboard area, to the left of the current ViewController. In the title bar of this table view controller, click the left most (yellow icon) and in the attributes inspector, choose “is initial View Controller”. Now control-drag from the Table View Controller’s yellow icon to the original view controller, to make a segue between the two of them. Note that we could have dragged from the cell to the original view controller to make a segue that would be automatically triggered when the user tapped on any cell in the table, but then we would not be able to do any processing before the segue is triggered, and you are going to need to do that later). For your non-automatic segue, choose “show” from the popup menu”. Once you’ve created the segue select it and in the attributes inspector name it “toDetail”. We will add code to trigger it later.

Add a new source file for the Table View Controller (currently it doesn’t have one). Go to the File menu and choose New and File…. Choose a Cocoa Touch class, click next, and on the options window, select “UITableViewController” from the subclass popup menu. Call the class ResearchPapersTVC (it’s a good idea to have an aspect of the class type in your name for an instance of a class such as this).


Choose options for your new file:

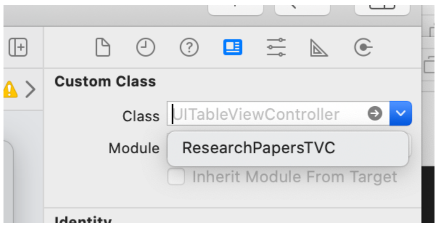
Class: ResearchPapersTVC

Subclass of: UITableViewController

Also create XIB file

Language: Swift

Click next and save the file to the project.
You’ll see that the file contains all the relevant methods for working with a table view. Now we need to connect that file to the Table View Controller on storyboard. Select that view controller in storyboard (click on its yellow icon) and using the Identity inspector, choose ResearchPapersTVC from the popup. We’re going to make an outlet from the table into our source, so that we can refresh the table etc. when we want to. Split the editor in the usual way, display the ResearchPapersTVC.swift file in one section and the main storyboard in the other, and control drag from the **table** into the ResearchPapersTVC class. Choose to create an outlet and call it theTable.


Custom Class

Class: UITableViewController

Module: ResearchPapersTVC

Inherit Module From Target

```
    @IBOutlet var theTable: UITableView!
```

Commit your work so far (Integrate menu, Commit option). In the commit message field, leave a comment such as “Initial setup of view controllers and associated class source file.” Click the commit button.

We are going to retrieve data from an API and display it in our table. The data is going to be in JSON format, so we’re going to use the codable feature (see from slide 33 to 36 of LS03 - Data Persistence part 2).

Click the link below to start up the Safari web browser, and display some JSON data from our research papers API.

https://cgi.csc.liv.ac.uk/~phil/Teaching/COMP228/techreports/data.php?class=techreports2&lastModified=2020-02-01%2000:00:00

In the browser window, you should see one item retrieved from the database via the API:

```
{
"techreports2": [
{"year":"2018","id":"1","owner":"tbc","email":"tbc@liverpool.ac.uk","authors":"Trevor
Bench-Capon, Katie Atkinson","title":"Relating the ANGELIC Methodology to
ASPIC+","abstract":"In this paper we relate the ANGELIC methodology for acquiring and
encapsulating domain knowledge to the ASPIC+ framework for structured argumentation. In
so doing we hope to facilitate the building of applications in concrete domains,
especially law, by linking a successful methodology to a proven theoretical framework. We
use an example from the ASPIC+ literature to illustrate the relationship, and as further
illustration provide an executable program able to support argument preparation. Finally
we identify several questions for future work.","pdf":"https:\/\/intranet.csc.liv.ac.uk\/
research\/techreports\/tr2018\/
ulcs-18-001.pdf","comment":"updated","lastModified":"2020-12-09 13:33:07"}]
}
```

It is difficult to see the structure, so I pasted the JSON text into the following web page:

https://jsonformatter.curiousconcept.com

Which validates and displays JSON in a nice human-readable way:

```
{
   "techreports2":[
      {
         "year":"2018",
         "id":"1",
         "owner":"tbc",
         "email":"tbc@liverpool.ac.uk",
         "authors":"Trevor Bench-Capon, Katie Atkinson",
         "title":"Relating the ANGELIC Methodology to ASPIC+",
         "abstract":"In this paper we relate the ANGELIC methodology for acquiring and
encapsulating domain knowledge to the ASPIC+ framework for structured argumentation. In
so doing we hope to facilitate the building of applications in concrete domains,
especially law, by linking a successful methodology to a proven theoretical framework. We
use an example from the ASPIC+ literature to illustrate the relationship, and as further
illustration provide an executable program able to support argument preparation. Finally
we identify several questions for future work.",
         "pdf":"https:\/\/intranet.csc.liv.ac.uk\/research\/techreports\/tr2018\/
ulcs-18-001.pdf",
         "comment":"updated",
         "lastModified":"2020-12-09 13:33:07"
      }
   ]
}
```

From this we can see that our API call returns a JSON dictionary (it’s inside of { } which indicates that we are working with a dictionary) containing one key - “techreports2”, whose value is an array (we can see that due to the [ ]) and each element of which is a dictionary. Each of those dictionaries contains the following keys:

`year`, **`id`**, `owner`, `email`, **`authors`**, **`title`**, `abstract`, `pdf`, `comment`, **`lastModified`**

The values of all of these are JSON strings, and some of them can be null (though not the ones in **bold**)

Now that we know the structure of the JSON data, we can make a Swift structure, or structures, that we can use with codable to help us to turn the JSON data into values within our program. Create another new file in your project, but this time, select a Swift file (not a Cocoa Touch class file), and called it **dataModel.swift** Put the following text into it (you can leave the comment lines at the top of the file).

```
import Foundation

struct techReport: Decodable {
    let year: String
    let id: String
    let owner: String?
    let email: String?
    let authors: String
    let title: String
    let abstract: String?
    let pdf: URL?
    let comment: String?
    let lastModified: String
}

struct technicalReports: Decodable {
    let techreports2: [techReport]
}
```

Note that our structures exactly match the names and types of JSON elements. If they did not, then decoding would not work (and is often the reason why this fails when writing API-based applications). Also, those items that can be null in the JSON must be defined as optionals in our structures.

Replace the viewDidLoad method in your ResearchPapersTVC.swift file with the following:

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

  if let url = URL(string: "https://cgi.csc.liv.ac.uk/~phil/Teaching/COMP228/
techreports/data.php?class=techreports2") {
      let session = URLSession.shared
        session.dataTask(with: url) { (data, response, err) in
          guard let jsonData = data else {
              return
          }
          do {
              let decoder = JSONDecoder()
              let reportList = try decoder.decode(technicalReports.self, from: jsonData)
              var count = 0
              for aReport in reportList.techreports2 {
                  count += 1
                  print("\(count) " + aReport.title) }
          } catch let jsonErr {
              print("Error decoding JSON", jsonErr)
          }
      }.resume()
      print("You are here!")
   }
}
```

(note that the URL string is on one line, not two as it appears here).

This code creates a URL, and then a URLSession for that URL. We then create a data task in which we retrieve the data from the URL (and deal with any error if it occurs). Once we have the data, we try to decode it from a JSON string into instance(s) of the appropriate structure(s) using decoder.decode( ). If it all works we will have the objects created from the decoded JSON in the reportList constant. We then access the techreports2 array from that, and iterate through the array elements to print the title part.

Commit your changes to the project. Run this app in the iPhone 15 simulator. Check the output in the console window in the lower section of Xcode. You should see details of all the papers printed out there.

Congratulations! You have managed to retrieve some data from an API. There's something important to notice though:

When you request data from a URL via a URLSession, the retrieval is done as a background task. The following line:

```
  session.dataTask(with: url) { (data, response, err) in
```

creates the start of a Swift *closure*, which is a self-contained block of code, which in this case is run asynchronously, since retrieval of data from a web service can take some time (up to 30 seconds!!). It could even time out if the server or network was not available. If this attempt to fetch the data occurred on the main process thread, then the app would freeze up until the data arrived. This is a bad idea and so that's why Swift provides this asynchronous method. The problem with this, is that the code inside the closure does not run in the order you may think it does, from a casual view of the source code listing. In fact, it is not executed at all until we reach the line

```
}.resume()
```

and if you look at the output in the console carefully (scroll it all the way back to the beginning), you will see that the message "You are here!" printed before any of the paper details were printed out - i.e. the print line was executed immediately (in the foreground) while the closure was starting to run in the background. (Note: you might, in addition, have some warning and diagnostic info in the console, which, for our purposes, you can ignore).

```
You are here!
1 On-line Dominating Sets in Web Graphs
2 Overview of CAT: A Market Design Competition
3 Verification for a Robotic Assistant
```

This running in a background thread thing is a problem, because you cannot put items into the table etc., until the data has arrived, and in iOS you can't do that in a background thread (Apple does not allow UI changes from a background thread). Also, the data we're retrieving and putting into reportList is only going to be available within the closure block. Once that's finished running it will go away and we won't be able to access our decoded papers. So what do we do?
Firstly, create a new property at the top of the ResearchPapersTVC class (inside the class)

```
    var reports:technicalReports? = nil
```

We're going to use that to get access to the decoded structure, from within the closure. Inside the closure put the following line on the line immediately after the try decoder.decode( ) line.

```
    self.reports = reportList
```

This will now point our class-level reports variable at the decoded data structure reportList. Now we will have access to the decoded data in the rest of our class, and it won't disappear once the closure finishes executing. There's still one more issue though. We can't update our table yet because we can't do it in the closure in the background.

Create a new method called updateTheTable, inside the ResearchPapersTVC class. For now we won't use it to update the Table, but instead to test that a method can be invoked from the background thread and that we're getting the data from the closure.

```
    func updateTheTable() {
        print(reports?.techreports2.count ?? 0)
    }
```

We need to invoke this method from our closure, which is running in the background, but we want this function to run in the foreground, not the background, and so we add DispatchQueue code to our closure, just after the line we added to copy the decoded structure to our reports variable. You can also now remove the code that printed out all the papers to the console. Your ViewDidLoad should now look like the following:

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

  if let url = URL(string: "https://cgi.csc.liv.ac.uk/~phil/Teaching/COMP228/
techreports/data.php?class=techreports2") {
      let session = URLSession.shared
      session.dataTask(with: url) { (data, response, err) in
          guard let jsonData = data else {
              return
          }
          do {
              let decoder = JSONDecoder()
              let reportList = try decoder.decode(technicalReports.self, from: jsonData)
              self.reports = reportList
              DispatchQueue.main.async {
                  self.updateTheTable()
              }
          } catch let jsonErr {
              print("Error decoding JSON", jsonErr)
          }
     }.resume()
  }
}
```

Try running your app again. This time in the console output you should see the number of items that were downloaded. This output was sent to the console from our updateTheTable( ) method, in the foreground!

Commit your project to save the changes you just made.

OK, so now we want to make the table display the relevant data. We need to setup our usual table delegate methods.

Edit the numberOfSections method to return 1 (rather than 0).
Edit the numberOfRowsInSection method to return either the number of items in the decoded array, or else 0. We can do this by using this code:

```
return reports?.techreports2.count ?? 0
```

Now we need to return the cell data.
Go to storyboard, select the table view cell, and use the attributes inspector to give it an identifier name - call it “theCell”. Now we can refer to that in our CellForRowAt indexPath method. Notice that that method is commented out in the code at the moment. Uncomment it and update the name of the reuse identifier to “theCell”. Add code to access the paper's title from our decoded structure. Your method should be as follows:

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "theCell", for: indexPath)
    var content = UIListContentConfiguration.cell()
    content.text = reports?.techreports2[indexPath.row].title ?? "no title"
    cell.contentConfiguration = content
    return cell
}
```

We either use the paper's title, or if for some reason the data is missing and the title is nil, we have supplied a default string of "no title".

Finally, we need to update our method that is called in the foreground from the closure, so that it redraws the table, once we have loaded the data that we need. Replace your updateTheTable method with the following:

```
func updateTheTable() {
    theTable.reloadData()
}
```

Run your app again. Your table should appear, and shortly afterwards it should be populated with all of the paper titles from the API. Pretty good eh? Note that clicking on one of the cells highlights it, but doesn't do anything else. Let's add some code to enable a tap in a cell to transition to the detail view controller, where you would display more info about the selected paper. What we need to do is to perform the segue that we setup earlier. And we need to do it when the cell is tapped on. For that we need a didSelect method, and in that do a performSegue. Add this code to your ResearchPapersTVC.

```
override func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
    performSegue(withIdentifier: "toDetail", sender: nil)
}
```

OK, run your app again. Tap on a cell, you should find that you transition to the second view controller. It doesn't display anything because you haven't programmed that yet. You'll have to swipe down to dismiss it and return to the table.

So our table contains the titles of the papers, but it would be better if we could also see the authors as well. We could do this by making a custom cell with appropriate UITextFields in it, but fortunately there's a simple change we can make to our cell properties on storyboard that will give it not one, but two text items and we can use those to display two pieces of information. In storyboard, select the cell, and then in the attributes inspector, select the style popup and choose "Subtitle". You will see that your cell now contains two text items: "Title" (the text property) and "subtitle" (the secondaryText property). Modify your CellForRowAt indexPath method. We need to change the kind of content we're creating (a subtitleCell not just a Cell) and add a line to set that secondary item of text to the authors string. Your cellForRowAt method should look like this:

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "theCell", for: indexPath)
    var content = UIListContentConfiguration.subtitleCell()
    content.text = reports?.techreports2[indexPath.row].title ?? "no title"
    content.secondaryText = reports?.techreports2[indexPath.row].authors ?? "no authors"
    cell.contentConfiguration = content
    return cell
}
```

Now run the app again. Even nicer now!!

Commit these recent changes to your project.

OK, for the additional points for this lab exercise:

- Setup the detail view controller to display the paper’s title, year, authors, email, abstract and the URL (only display this if it is not nil). Make the URL clickable - so that it opens safari to show you the url contents. Hint: a textView (rather than a label) can help with this.

- Pass the data across using the segue (see slides 92 & 93 of LS02 - iOS Fundamentals part 4) - do not use global variables to do this.

Remember to make commits to your Git repository (ideally after you’ve implemented each additional step - with a little comment about what you did each time).

Please Note:
This is the final one of the lab exercises for your portfolio. There will be some other lab work which will be of use to you in assignment two, but no more apps for part two of your portfolio. It is all due in at the end of week 12, but you may wish to submit it as soon as you have finished, in order to leave plenty of time for assignment two. Part two of the portfolio consists of the following three items:

Week 6: Core Data exercise
Week 8: My Favourite Places
Week 9: Research Papers