

Lab 5 – Working with Web Services

Overview:

For the lab task this week you will be leveraging web services via a REST API. Using online services is very common in application development, whether the API is custom created or public. This lab examines the Google Books API as an example of such usage.

This lab assumes knowledge of Core Data from the previous lab exercise.

Note: Some of the Macs in the lab may be running Windows and need to be rebooted into the OS X environment. Restart the Mac and hold down the Option key while it is booting. You will need to select 'EFI Boot' and select OS X to launch the correct operating system.

The Task:

In this lab you will create a personal book collection app. The app will leverage Core Data to store and display a list of books representing the user's personal library. The app has a second screen that allows the user to query the Google Books API with a book name and add a book from the returned results.

This will require the following tasks:

- Create JSON data classes
- Create the Search Books Table View Controller
- Create the My Books Table View Controller
- Create a Core Data Model
- Create a Database Listener
- Create a Database Protocol
- Create a Core Data Controller Class

Creating a new project:

Start by creating a new Storyboard-based UIKit App project, as in previous weeks.

Creating the JSON Data Classes:

For this week's task we will use the Decodable protocol to automatically decode the JSON data from Google's API into data classes, before they are added to Core Data. Only books that the user selects will be made into Core Data objects to be saved.

First, let's examine the JSON returned from the Google API query. Copy the contents of <https://www.googleapis.com/books/v1/volumes?q=Unreal%20Engine> into the JSON Beautifier website at <https://codebeautify.org/jsonviewer> and click the "Tree Viewer" button.

The base of the returned information is a JSON object with 3 fields; kind, totalItems and items.

Kind and Total Items are not important to what we are doing, so we can ignore those.

However, we will need to get access to the list of items. These are the books.

Each item represents a single book object.

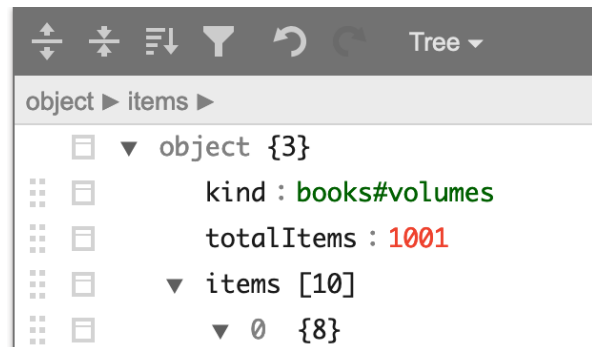
It contains information relating to both the book itself as well as further information for interacting with the book via the API.

You may notice the Book information we want is actually nested down within “volumeInfo”. We are going to have to manually map our class a little bit

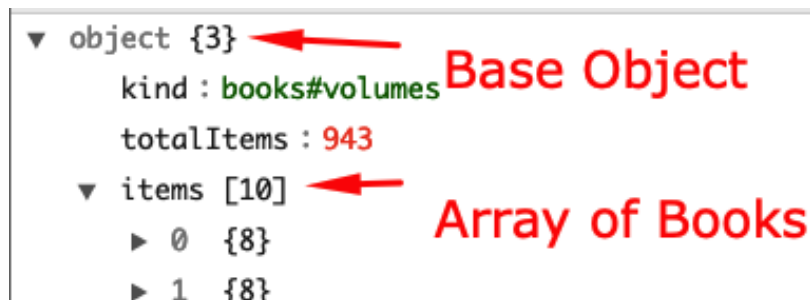
The hierarchy for a book is as follows:

Item

- volumeInfo
 - o title
 - o authors (array)
 - o publisher
 - o publishedDate
 - o description
 - o industryIdentifiers (array)
 - type
 - identifier
 - o imageLinks
 - smallThumbnail



We will create two classes to map this information. The first will correspond to the query result object and will be a container for holding a list of books. This is required because the base result returned from the JSON query is an object, not an array of books.



To begin, create a new class called “VolumeData” ensuring that it inherits from **NSObject** and **Decodable**.

Create a second class called “BookData” ensuring that it inherits from NSObject and Decodable. We need to create both quickly because VolumeData will need to refer to this class.

VolumeData

Give the VolumeData class a property named “books”, an optional array of BookData instances:

```
var books: [BookData]?
```

CodingKeys defines the mapping between the object and the format being decoded from (i.e., the JSON data). Apple has a fantastic breakdown of this process here: https://developer.apple.com/documentation/foundation/archives_and_serialization/encoding_and_decoding_custom_types.

For this class create a CodingKeys property to correctly map our books property to the items array in the JSON format:

```
private enum CodingKeys: String, CodingKey {  
    case books = "items"  
}
```

This is all that is needed for VolumeData as the books array is the only information we need from it.

BookData

The first step when creating the BookData class is understanding the properties that we will want to have. At a minimum we should include the following for this kind of application:

- ISBN-13
- Title
- Authors
- Publisher
- Publish Date
- Book Description
- Image URL

The next step of the process is to determine appropriate property types. To do this, look at the JSON response data and check the data types.

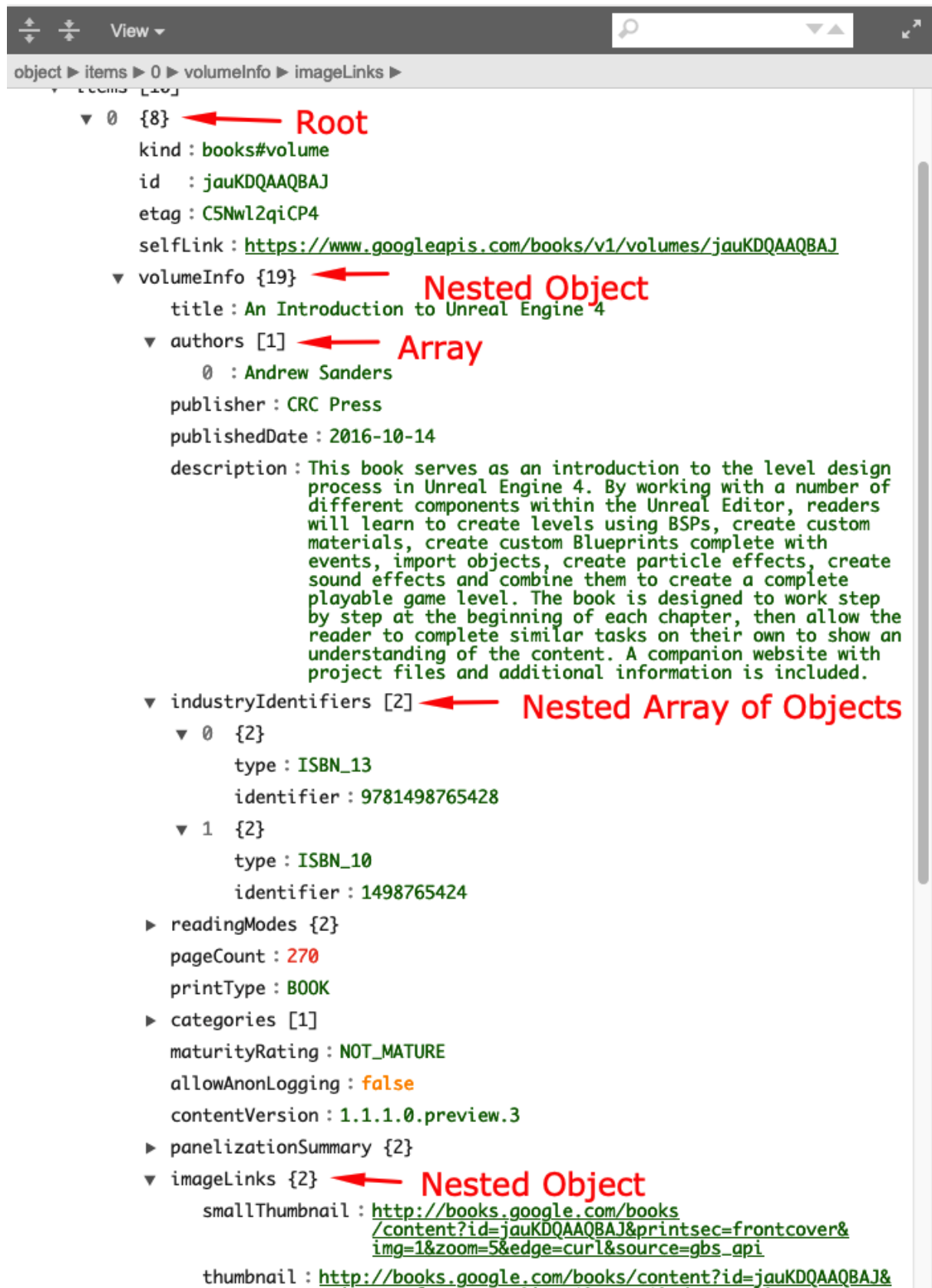
```
{
  "kind": "books#volumes",
  "totalItems": 943,
  "items": [
    {
      "kind": "books#volume",
      "id": "jauKDQAAQBAJ",
      "etag": "CSNwL2qiCP4",
      "selflink": "https://www.googleapis.com/books/v1/volumes/jauKDQAAQBAJ",
      "volumeInfo": {
        "title": "An Introduction to Unreal Engine 4",
        "authors": [
          "Andrew Sanders"
        ],
        "publisher": "CRC Press",
        "publishedDate": "2016-10-14",
        "description": "This book serves as an introduction to the level design process in Unreal Engine 4. By working with a number of different components within the Unreal Editor, readers will learn to create levels using BSPs, create custom materials, create custom Blueprints complete with events, import objects, create particle effects, create sound effects and combine them to create a complete playable game level. The book is designed to work step by step at the beginning of each chapter, then allow the reader to complete similar tasks on their own to show an understanding of the content. A companion website with project files and additional information is included.",
        "industryIdentifiers": [
          {
            "type": "ISBN_13",
            "identifier": "9781498765428"
          },
          {
            "type": "ISBN_10",
            "identifier": "1498765424"
          }
        ],
        "readingModes": {
          "text": true,
          "image": true
        }
      }
    }
  ],
}
```

Fortunately, all the properties we care about are Strings. However, we also need to decide if each property is optional. This comes down to whether the information is always available for a book or not. It can be determined by trial and error or by looking at API documentation.

In this case the title will always have a value in the JSON response. Everything else is optional. As such, our final set of properties should look as follows:

```
var isbn13: String?
var title: String
var authors: String?
var publisher: String?
var publicationDate: String?
var bookDescription: String?
var imageURL: String?
```

With properties decided, the next step is creating the coding keys. However, unlike volume data the process for a book is much more involved. Looking at the Tree View of the JSON data you may notice that a single book contains multiple nested JSON objects and arrays.



To solve this issue there are a few options. One option would be to create a separate class for each of the nested objects. While that would be simpler, we would prefer to distill the book data down to a single object.

For this approach, we can create multiple sets of Coding Keys for each nested object and implement our own version of the initializer required by the decodable protocol.

To begin, we will create the root coding keys. This is our root object as identified on the previous page. The only thing we need to access within it is the "volumeInfo" nested JSON object. As such the Root Keys only includes this:

```
private enum RootKeys: String, CodingKey {  
    case volumeInfo  
}
```

This is our first layer that allows us to get access to the nested volumeInfo object. The volumeInfo object contains most of the information that we need as properties, with the exception of the authors, imageURL and ISBN-13 which are in nested arrays or objects. Create CodingKeys for the volumeInfo (Book):

```
private enum BookKeys: String, CodingKey {  
    case title  
    case publisher  
    case publicationDate = "publishedDate"  
    case bookDescription = "description"  
    case authors  
    case industryIdentifiers  
    case imageLinks  
}
```

Where the case matches our property names we do not need to specify it with a value. Only in instances where they differ—such as publicationDate and bookDescription (named so because NSObject has a builtin description method)—are we required. All properties that we wish to obtain from the JSON data MUST be specified.

Next is a CodingKey to access the imageURL. The image URL is contained within the nested imageLinks object that has two properties; smallThumbnail and thumbnail. Add this:

```
private enum ImageKeys: String, CodingKey {  
    case smallThumbnail  
}
```

The last thing that is needed is a struct class for the ISBN. ISBN objects have a type and identifier. As we need both values to be able to determine we have the correct ISBN code it is easier to just create it as a struct. As with the enums, this is defined within the class.

```
private struct ISBNCode: Decodable {  
    var type: String  
    var identifier: String  
}
```

With the Coding Keys defined, we can implement the initializer.

```
required init(from decoder: Decoder) throws {  
}
```

There are a few noticeable differences with this initializer than ones we have created previously. This is passed a decoder class instance. The other difference is that this initializer can throw an error.

Inside of the initializer we must first create a container for the root JSON object then one for volume data and images respectively.

```
// Get the root container first  
let rootContainer = try decoder.container(keyedBy: RootKeys.self)  
// Get the book container for most info  
  
let bookContainer = try rootContainer.nestedContainer(keyedBy:  
    BookKeys.self, forKey: .volumeInfo)  
  
// Get the image links container for the thumbnail  
let imageContainer = try? bookContainer.nestedContainer(keyedBy:  
    ImageKeys.self, forKey: .imageLinks)
```

For the **rootContainer** we only need to provide the Coding Keys that it is using.

For both the **bookContainer** and **imageContainer** we use the **nestedContainer** method instead and must provide both the Coding Keys it will use as well as its key within the parent container.

From the bookContainer we can easily get the title, publisher, publicationDate and bookDescription with minimal effort:

```
// Get the book info
title = try bookContainer.decode(String.self, forKey: .title)
publisher = try? bookContainer.decode(String.self, forKey: .publisher)
publicationDate = try? bookContainer.decode(String.self, forKey:
    .publicationDate)
bookDescription = try? bookContainer.decode(String.self, forKey:
    .bookDescription)
```

When decoding, the type is required along with the key. A decode attempt can fail which is why the try keyword is used, though a do-catch statement is not present. That is because it is handled by whichever code attempts to decode a book! For optional properties that may or not be present, a “try?” statement is used, since if that decode fails it just results in a value of nil for the property.

Getting authors of a book requires a little more work. As it is an array, we must decode it as an array. We check if there were authors in the JSON object. If so, we convert the array of strings to a single string using the array joined method.

```
// Get authors as an array then compact
if let authorArray = try? bookContainer.decode([String].self, forKey:
    .authors) {

    authors = authorArray.joined(separator: ", ")
}
```

Getting the ISBN objects is more or less the same process. Instead of creating a String array from the decode we instead create an array of ISBNCode objects (defined by our struct). From here if we can find an ISBNCode with the type "ISBN_13" we store it.

```
// ISBN 13 takes a little more work to get done
// First get the ISBNCodes as an array of ISBNCode
if let isbnCodes = try? bookContainer.decode([ISBNCode].self, forKey:
    .industryIdentifiers) {

    // Loop through array and find the ISBN13
    for code in isbnCodes {
        if code.type == "ISBN_13" {
            isbn13 = code.identifier
        }
    }
}
```


The final step for the initializer is to get the imageURL. Thankfully this is another simple one as we have a container to quickly grab the data

```
// Lastly get the image thumbnail from the imageContainer  
imageURL = try imageContainer?.decode(String.self, forKey: .smallThumbnail)
```

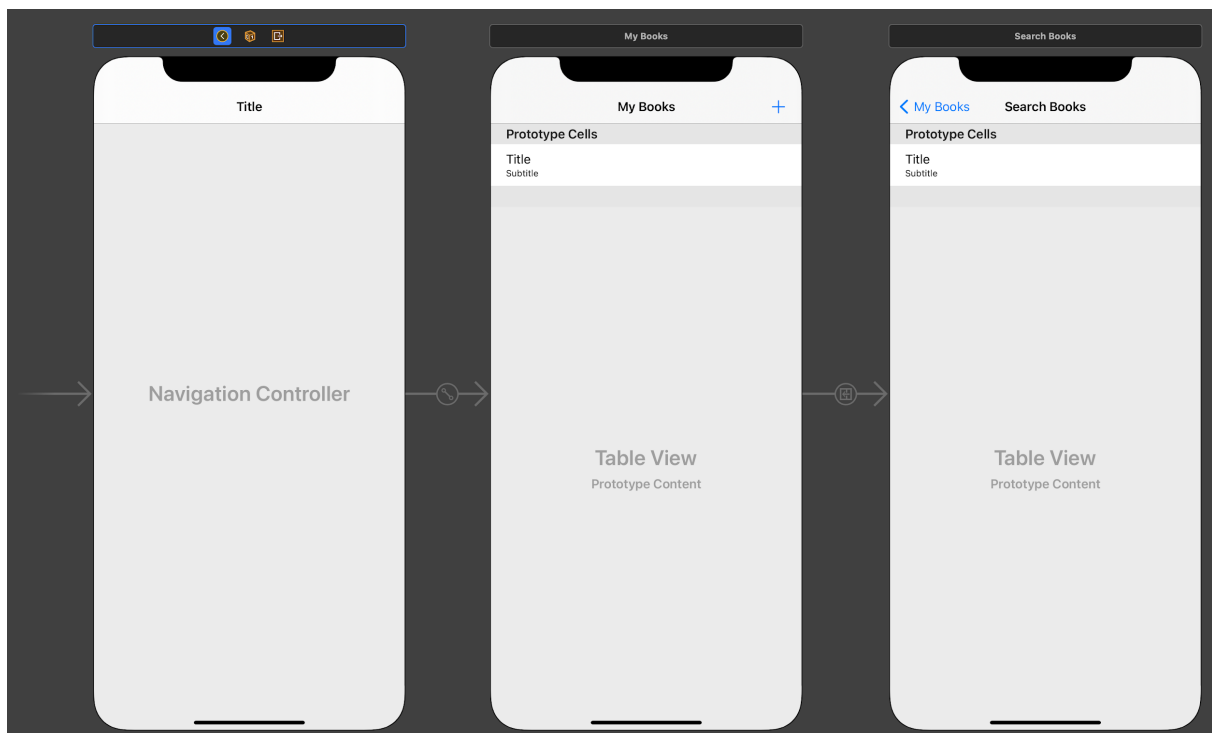
With this, the BookData class is complete.

If you have difficulty understanding the decoding of the JSON by this class, once you have finished the rest of the steps you should set a breakpoint in the BookData init method and watch the result of each decode line.

Creating the Storyboard:

The Storyboard for this application is fairly straightforward, with a navigation view and two table view controllers. Unlike previous weeks we will be creating and testing the second screen first, which is the Search Books Table View Controller.

Create the storyboard so that it looks like the image provided below. For the segue between the "My Books" screen and the "Search Books" screen you should ensure that it is triggered by the Add button in the top right corner of the screen.



For the first table view controller, create a new UITableViewController class called "MyBooksTableViewController" and ensure that it is linked. Set the reuse identifier for the prototype cell to be "bookCell" and set its style to "Subtitle".

For the second table view controller create a new UITableViewController class called “SearchBooksTableViewController” and ensure that it is linked. Again, set the cell identifier for the prototype cell to be “bookCell” and set its style to “Subtitle”.

Coding the SearchBooksTableViewController:

We will begin by building and testing the second screen first. Open the SearchBooksTableViewController. Ensure that it adopts the UISearchBarDelegate protocol:

```
class SearchBooksTableViewController: UITableViewController,
UISearchBarDelegate {
```

Next, we need to set up our class properties. As always, a constant for the cell identifier is important. We will also declare a constant for the API URL.

```
let CELL_BOOK = "bookCell"
let REQUEST_STRING = "https://www.googleapis.com/books/v1/volumes?q="
```

The class will also need an array of BookData objects for displaying to the user:

```
var newBooks = [BookData]()
```

The last class property we need is an Activity Indicator View. This is a UI element that displays a spinning animation most commonly used to indicate loading. We will be handling this programmatically as it is easier to hide and unhide based on the search and web service response:

```
var indicator = UIActivityIndicatorView()
```

With that done we can move onto coding the table view data source delegate methods. This week we only need 3 and they are all simple implementations.

numberOfSections

The table view will have one section.

numberOfRowsInSection

Set the return value to be the side of the newBooks array size (count).

cellForRowAtIndexPath

Dequeue a regular cell using the CELL_BOOK identifier. Get the specified book from the newBooks array and set the cell labels with the book title and authors:

```
let book = newBooks[indexPath.row]
cell.textLabel?.text = book.title
cell.detailTextLabel?.text = book.authors
```

viewDidLoad

With the TableView methods complete we can move onto filling out the required information in viewDidLoad. Firstly, we need to create a Search Controller and attach it to our navigation item.

```
let searchController = UISearchController(searchResultsController: nil)
searchController.searchBar.delegate = self
searchController.obscuresBackgroundDuringPresentation = false
searchController.searchBar.placeholder = "Search"
navigationItem.searchController = searchController

// Ensure the search bar is always visible.
navigationItem.hidesSearchBarWhenScrolling = false
```

This matches Lab 3. However, this week is that we will be implementing a different method as part of the Search Bar Delegate.

Finally for viewDidLoad, we need to set up and add our indicator to the view controller's view. First, we set the style of the indicator, turn off automatic constraints, and programmatically use constraints to centre it in the safeArea of the view:

```
// Add a loading indicator view
indicator.style = UIActivityIndicatorView.Style.large
indicator.translatesAutoresizingMaskIntoConstraints = false
self.view.addSubview(indicator)

NSLayoutConstraint.activate([
    indicator.centerXAnchor.constraint(equalTo:
        view.safeAreaLayoutGuide.centerXAnchor),
    indicator.centerYAnchor.constraint(equalTo:
        view.safeAreaLayoutGuide.centerYAnchor)
])
```

requestBooks

Create a method called `requestBooksNamed` with an anonymous parameter `bookName` (a `String`). This is the method where we will call the API to request book information. We will fill this out shortly.

```
func requestBooksNamed(_ bookName: String) {  
  
}
```

searchBarTextDidEndEditing

The next method we need to implement is the `searchBarTextDidEndEditing` method from the `UISearchBarDelegate` protocol. This is called if the user hits enter or taps the search button after typing in the search field. It is also called when the user taps cancel.

```
func searchBarTextDidEndEditing(_ searchBar: UISearchBar) {  
  
}
```

When the method is called, we should empty the current list of `newBooks` and refresh the `tableView`, as a new search is about to begin:

```
newBooks.removeAll()  
tableView.reloadData()
```

Next, we should guard against the search bar text being nil or being empty. In either of these cases we should return immediately as there is no point trying to query the API. Write a guard statement to achieve this.

If there is search text we should make the indicator appear to provide feedback that the application will be loading something that may not finish instantly. Lastly, we should call the `requestBooks` method to handle the request:

```
indicator.startAnimating()  
requestBooksNamed(searchText)
```

requestBooks

This method is responsible for making a request to the API, receiving results and parsing them into a usable format. There are a couple new concepts here primarily using background tasks and communicating across threads to send information back to the main dispatch queue. For more information on the topic check out <https://developer.apple.com/documentation/dispatch/dispatchqueue>.

Before executing a query we must first create the URL for the API request.

```
guard let queryString = bookName.addingPercentEncoding(withAllowedCharacters:
    .urlQueryAllowed) else {
    print("Query string can't be encoded.")
    return
}

guard let requestURL = URL(string: REQUEST_STRING + queryString) else {
    print("Invalid URL.")
    return
}
```

The requestURL is a combination of the request string constant property plus the search text entered by the user (encoded to use characters safe for URL query strings).

Next, we need to create the data task and execute it. This is relatively simple but it is important to note the behaviour of the completion handler.

```
let task = URLSession.shared.dataTask(with: requestURL) {
    (data, response, error) in

    // This closure is executed on a different thread at a later point in
    // time!
}

task.resume()
```

That task is passed a completion handler closure that will execute when the data task has received (or failed to) receive a response from the web API. We have no control over when the completion handler will be called.

INSIDE OF THE CLOSURE

When the closure is called, the first thing we want to do is to tell our loading indicator to stop animating (by default it also hides when not animating). At this point we have finished loading and will be ready to display results.

```
DispatchQueue.main.async {
    self.indicator.stopAnimating()
}
```

As the closure is running on a separate queue, but all changes to the UI must be made from the main thread, will call the indicator `stopAnimating` method via a `DispatchQueue.main.async` block.

The next step of the closure is to check if we have received an error message. If this is the case it means something has gone wrong and we shouldn't attempt to parse results. In this case we can just print the error to the console for debugging purposes and return immediately.

```
if let error = error {  
    print(error)  
    return  
}
```

After this point, we know there was no error and we should be able to parse the data. Parsing data through a decoder can throw an error so we must use a `do/catch` block to handle this.

```
do {  
    let decoder = JSONDecoder()  
    let volumeData = try decoder.decode(VolumeData.self, from: data!)  
  
    if let books = volumeData.books {  
        self.newBooks.append(contentsOf: books)  
  
        DispatchQueue.main.async {  
            self.tableView.reloadData()  
        }  
    }  
} catch let err {  
    print(err)  
}
```

First, we create a `JSONDecoder` instance, which can make use of our `Codeable` object. More information can be found on the decoder [here](https://developer.apple.com/documentation/foundation/jsondecoder).

Once we have the decoder we attempt to decode the `volumeData` from the data obtained from the request. If this fails the catch statement will print the error.

If the `volumeData` was successfully decoded, we check that the `books` property is not nil. If this is the case, we append the retrieved books to our property `newBooks` and reload the tableview. At this point we have displayed the books from the search query!

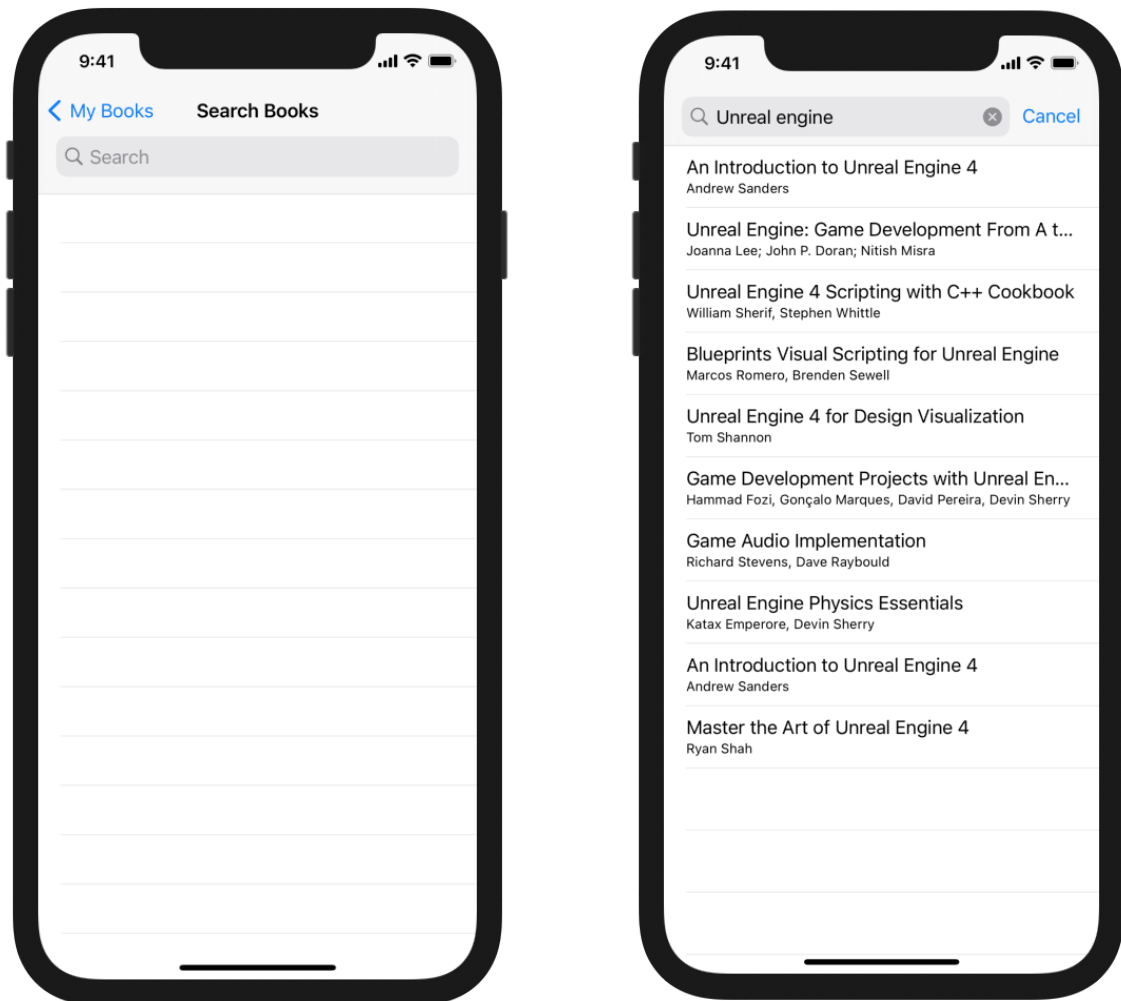
Did you know?

You might have noticed that we used append when getting our searched books and putting them into our newBooks array. Why didn't we just assign books to the newBooks property?

The API query we are using is paged, which means it only returns the first 40 results! In a later step we will be calling the query repeatedly to fetch all pages of data!

Testing the Application:

Whilst we have not implemented the book saving functionality, it is a good idea to test the application at this point to ensure that the search and API code works. Run the application, go to the second screen and enter a search term (Unreal Engine is used in the example below). You should be able to see a loading indicator followed by a list of matching results. Our searching is working correctly!



Creating the Core Data stack:

With the search functionality working, it is time to create the Core Data stack so that we can save our books to persistent memory. This week will be a simpler data model as there is less information to store.

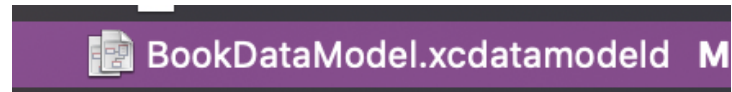
Create a new Data Model named “BookDataModel”.








Create a new entity called Book with the following attributes, all of type String:

- authors
- bookDescription
- imageURL
- isbn13
- publicationDate
- publisher
- title

Set the entity’s Codegen to “Manual/None” in the Data Model inspector.

Create an NSObject Subclass for the Book entity.



Attribute	Type
 title	String
 authors	String
 bookDescription	String
 imageURL	String
 isbn13	String
 publicationDate	String
 publisher	String
+	—

As with Lab 4, add the MulticastDelegate.swift file to the project.

Create a new Swift file called “DatabaseProtocol”. In this we will define our DatabaseListener and DatabaseProtocol like last week. This time however, they are simpler implementations.

Define a DatabaseListener protocol that inherits from AnyObject. It should have a single method called **onBookListChange**, with a parameter “bookList”, an array of Book instances.

Define a DatabaseProtocol protocol that inherits from AnyObject. It should have a method for adding and removing a listener, adding a book and cleanup. The addBook should have a parameter “bookData” of type BookData and return a Core Data Book.

With this done, it is time to create the CoreDataController itself. Create a new class called “CoreDataController” ensuring that it inherits from NSObject, NSFetchedResultsControllerDelegate and DatabaseProtocol.

Properties

As with last week, the CoreDataController requires a few properties.

```

var listeners = MulticastDelegate<DatabaseListener>()
var persistentContainer: NSPersistentContainer
var allBooksFetchedResultsController: NSFetchedResultsController<Book>?

```

Initializer

For the initializer we need to load the persistentContainer in the same fashion as Lab 4.

```

persistentContainer = NSPersistentContainer(name: "BookDataModel")
persistentContainer.loadPersistentStores() { (description, error) in
    if let error = error {
        fatalError("Failed to load Core Data stack with error: \(error)")
    }
}

super.init()

```

fetchAllBooks method

Create a method called fetchAllBooks that returns an array of books. This method will be similar to the fetchAllHeroes method from last week's lab.

```

if allBooksFetchedResultsController == nil {
    let fetchRequest: NSFetchedRequest<Book> = Book.fetchRequest()
    let nameSortDescriptor = NSSortDescriptor(key: "title", ascending: true)
    fetchRequest.sortDescriptors = [nameSortDescriptor]

    allBooksFetchedResultsController = NSFetchedResultsController<Book>(
        fetchRequest: fetchRequest, managedObjectContext:
        persistentContainer.viewContext, sectionNameKeyPath: nil,
        cacheName: nil)

    allBooksFetchedResultsController?.delegate = self

    do {
        try allBooksFetchedResultsController?.performFetch()
    } catch {
        print("Fetch Request failed: \(error)")
    }
}

if let books = allBooksFetchedResultsController?.fetchedObjects {
    return books
}

return [Book]()

```

addListener method

The addListener method is again almost identical to last week:

```
func addListener(listener: DatabaseListener) {
    listeners.addDelegate(listener)
    listener.onBookListChange(bookList: fetchAllBooks())
}
```

removeListener method

The removeListener method is completely identical to last week

```
func removeListener(listener: DatabaseListener) {
    listeners.removeDelegate(listener)
}
```

addBook method

The addBook method is similar to the structure of the addSuperHero method from last week. The main difference here is that a BookData object should be passed in and a Book Core Data entity should be returned.

```
func addBook(bookData: BookData) -> Book {
    let book = NSEntityDescription.insertNewObject(forEntityName: "Book",
        into: persistentContainer.viewContext) as! Book

    book.authors = bookData.authors
    book.bookDescription = bookData.bookDescription
    book.imageURL = bookData.imageURL
    book.isbn13 = bookData.isbn13
    book.publicationDate = bookData.publicationDate
    book.publisher = bookData.publisher
    book.title = bookData.title

    return book
}
```

cleanup method

The cleanup step should save the managed object context if there are pending changes. This will always be called when the application is shutting down.

```
func cleanup() {
    if persistentContainer.viewContext.hasChanges {
        do {
            try persistentContainer.viewContext.save()
        } catch {
            fatalError("Failed to save data to Core Data with error \(error)")
        }
    }
}
```

controllerDidChangeContent method

Lastly, the didChangeContent method should invoke each listener and call its onBookListChange method. This week due to the fact we only have a single controller, we do not need to have additional checks in here.

```
func controllerDidChangeContent(_ controller:
    NSFetchedResultsController<NSFetchRequestResult>) {

    listeners.invoke() { listener in
        listener.onBookListChange(bookList: fetchAllBooks())
    }
}
```

This completes the CoreDataController. Before making any changes to our search page we need to initialise the controller within the App Delegate and save changed within the SceneDelegate.

AppDelegate class

```
var databaseController: DatabaseProtocol?

func application(_ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions:
    [UIApplication.LaunchOptionsKey: Any]?) -> Bool {

    databaseController = CoreDataController()
    return true
}
```

SceneDelegate class

```
func sceneDidEnterBackground(_ scene: UIScene) {  
    (UIApplication.shared.delegate as?  
        AppDelegate)?.databaseController?.cleanup()  
}
```

Saving Books from Search:

Now that we have a persistent database, we can add code to save books when they are selected in the search screen. Return to the SearchBooksTableViewController and make the following changes.

Add a databaseController property to the class:

```
weak var databaseController: DatabaseProtocol?
```

In viewDidLoad get a reference to the database from the appDelegate:

```
let appDelegate = (UIApplication.shared.delegate as? AppDelegate)  
databaseController = appDelegate?.databaseController
```

Lastly, implement the didSelectRowAt method provided by the UITableViewController. In this method we will get the selected book from the TableView and pass it into the database controller. We then pop the search scene off the navigation stack.

```
override func tableView(_ tableView: UITableView, didSelectRowAt  
    indexPath: IndexPath) {  
  
    let book = newBooks[indexPath.row]  
    let _ = databaseController?.addBook(bookData: book)  
    navigationController?.popViewController(animated: true)  
}
```

This is all that is needed to save books to Core Data. It is a good idea to test the application at this point. However, you may notice that there are no results shown on the My Books screen yet. And that is because we need to implement the functionality.

Creating the MyBooksTableViewController:

Open the MyBooksTableViewController swift file. Ensure that it adopts the DatabaseListener protocol.

For class properties ensure that it has the following:

- `CELL_BOOK` constant to hold the identifier for the prototype cell (identical to Search Screen)
- `allBooks` property that is an array of Book objects
- A weak var property `databaseController`, an Optional DatabaseProtocol

As with the `SearchBooksTableViewController`, for **`viewDidLoad`** method ensure that the `databaseController` is set up by getting a reference from the `AppDelegate`.

Create both an overridden method for **`viewWillAppear`** and **`viewWillDisappear`** in the same fashion as last week. Ensure that they add and remove the current class as a listener respectively.

For **`numberOfSections`** return 1.

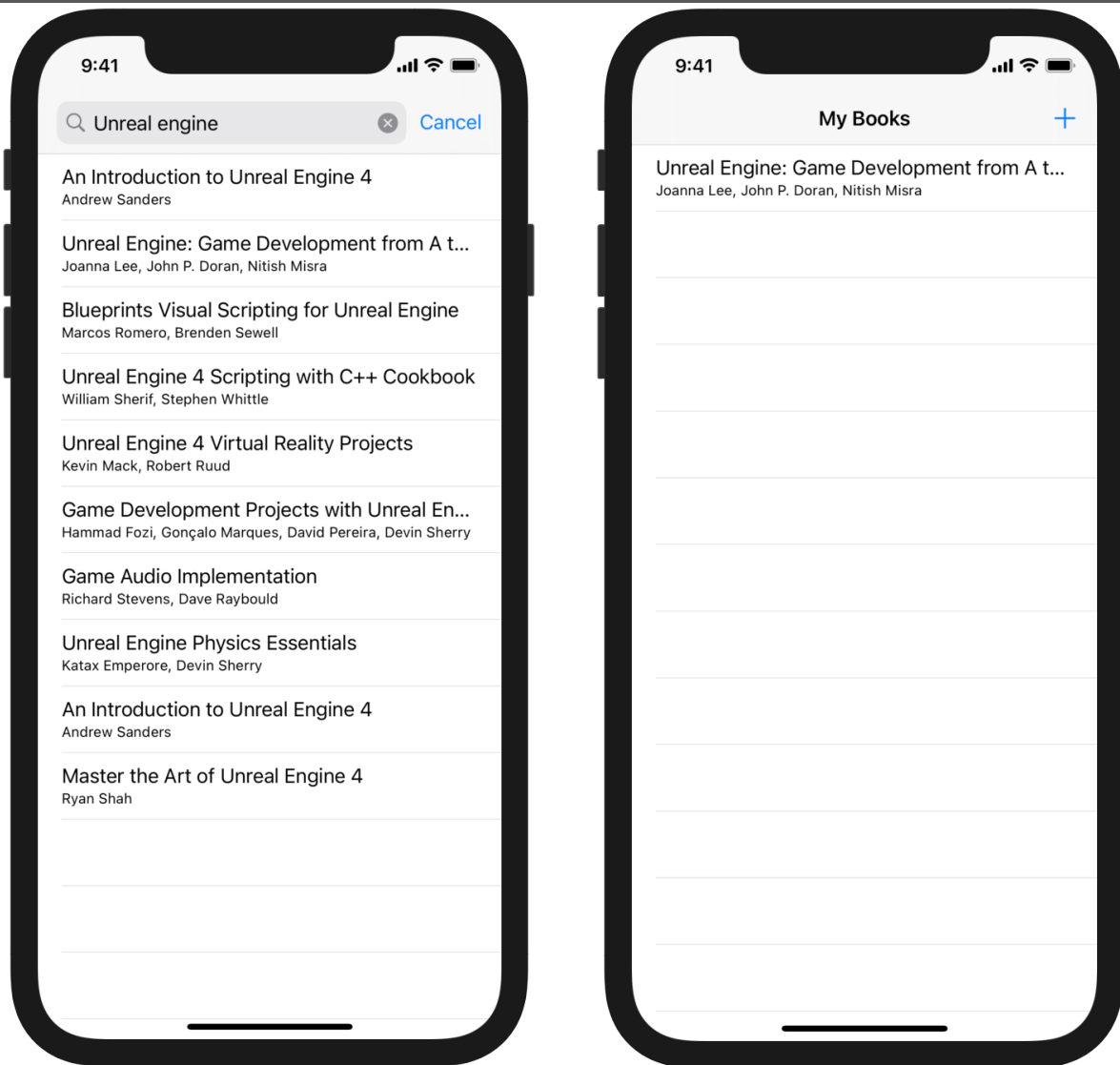
For **`numberOfRowsInSection`** return the size of the `allBooks` array

For the **`cellForRowAt`** method, set it up in the same fashion as the Search Screen. The only difference will be `allBooks` instead of `newBooks`.

Lastly implement the **`onBookListChange`** method as required by the `DatabaseListener` protocol. (Set `allBooks` to be the given `bookList` and reload the `tableView`.)

Testing the Application:

With these additions the application is almost complete. Test the application and you should now be able to search for books and have them save and appear on the My Books Screen. Remember that the books are only saved if the app is closed on the simulator not stopped by Xcode.



Downloading all Results:

If you have looked at book search results in a web browser you may have noticed that there are way more than 40 results for most search terms. The Google Books API allows you to request at most 40 books per search request, meaning we cannot do a single request to get all results.

However, we can perform multiple requests to get all search results. Let's make the necessary changes to the SearchBooksTableViewController.

Firstl, add the following additional class properties.

```
let MAX_ITEMS_PER_REQUEST = 40
let MAX_REQUESTS = 10

var currentRequestIndex: Int = 0
```

It is good practise not to use magic numbers. We need constants for the maximum items that can be requested and the maximum requests.

In order to know where we need to start the search query, we keep track of the current index in the request.

Next inside of the `searchBarTextDidEndEditing` method add the following two lines **before** the `requestBooksNamed` method call.

```
URLSession.shared.invalidateAndCancel()
currentRequestIndex = 0
```

This is still only called once when the search button is tapped. We assume that when this happens, we want to start a new search.

Calling `invalidateAndCancel` on the URL session forcefully stops all existing tasks. Without this, previous search results will still continue to download in the background. With the task cancelled, we reset the `currentRequestIndex` back to 0.

Finally, within the `requestBooks` method replace the two guard statements with the following:

```
var searchURLComponents = URLComponents()
searchURLComponents.scheme = "https"
searchURLComponents.host = "www.googleapis.com"
searchURLComponents.path = "/books/v1/volumes"
searchURLComponents.queryItems = [
    URLQueryItem(name: "maxResults", value: "\(MAX_ITEMS_PER_REQUEST)"),
    URLQueryItem(name: "startIndex", value: "\(currentRequestIndex *
        MAX_ITEMS_PER_REQUEST)"),
    URLQueryItem(name: "q", value: bookName)
]

guard let requestURL = searchURLComponents.url else {
    print("Invalid URL.")
    return
}
```

The `URLComponents` class allows us to neatly build URLs from its various components and neatly set up query items. Previously, this mattered less as we only had a single query item, the search text. However, as we need to call this method repeatedly until all books have been loaded through the pages, we need to specify how many items we plan to request in each page and what page we are currently on.

We add a property `maxResults` to the query string. We specify that we want up to 40 results each request.

More properties are found at:

<https://developers.google.com/books/docs/v1/reference/volumes/list>

We also specify `startIndex`. This is used to offset the results and get more than the first 40 results.

The final change needed to this class is to ensure that when the books finish loading from the API if there are more pages remaining we need to call the method again after increasing the request count index.

Add the following code under the `DispatchQueue.main.async` call for reloading the `tableView`.

```
DispatchQueue.main.async {  
    self.tableView.reloadData()  
}  
  
if books.count == self.MAX_ITEMS_PER_REQUEST,  
    self.currentRequestIndex + 1 < self.MAX_REQUESTS {  
  
    self.currentRequestIndex += 1  
    self.requestBooksNamed(bookName)  
}
```

This method will now be called repeatedly for each batch of 40 results, until there are no results remaining.

This will happen in the background but will happen sequentially. Only one request will be in progress at once.

After adding results, we check to see if the new book count is equal to the number of results we requested, i.e., this search request returned the maximum number of results and there might be more. If it is and we haven't performed the maximum number of requests, we then increment the `currentRequestIndex` by the max request number and call the method again. If not, then we are at the end of the results so do nothing!

Testing the Application:

At this point you can run the application and enter a search term. If there are more than 40 results you will notice the `tableView` being constantly updated until there are no more results!

What next?

There are a lot of potential improvements you could make to this app.

See if you can complete some of the following:

- Add in a screen to view more data on the book. Currently the data is in Core Data but not shown in the app!
- Extend the app so books can be removed as well as added to the My Books view.
- Change the cells in each screen to be custom and stretch to fit the book name. The book name is truncated if it is too long. Can you make the cells size to fit the name? (Hint: UILabels can have multiple lines.)
- Change the Add Book Screen to allow the user to select multiple books at once.
- Add the image thumbnail to each cell. Each book currently stores a URL for an image thumbnail. Can you get it to download and display in the cell?
- Whilst our downloading of requests works. It is not the recommended method of fetching additional results from an API which could return thousands of results. See the following tutorial for how to implement the prefetching API to fetch additional results as the user scrolls:

<https://www.raywenderlich.com/5786-uitableview-infinite-scrolling-tutorial>