

Minitutorial 8

Realización de búsquedas y filtrados



Alfonso López Ruiz



Constantes (let)

Variables inmutables.

```
let cellIdentifier = "MealTableViewCell"
```

Arrays

Conjunto lineal de elementos accesibles a través de sus índices.

Utilizaremos especialmente los métodos **remove** (actualización y borrado) e **index(of:)** para identificar el índice de una comida en un vector.

También usaremos el operador **+=** para añadir un vector a otro y la propiedad **count** para conocer su tamaño.

```
searchController.searchBar.scopeButtonTitles = ["Name", "Rating"]
```

Funciones

Bloques de código con un propósito muy concreto.

Nos interesa especialmente el uso de la palabra reservada **private** para definir funciones privadas, así como los **valores por defecto de los argumentos** de una función.

```
private func filterContentForSearchText(..., scope: String = "Name")  
{  
  
}
```

Variables opcionales (Optionals)

Variables que pueden contener un valor o no (nil).

```
var newPath : IndexPath?
```

Obtener valor de variable opcional (Unwrapping)

Se hará mediante el símbolo !

```
searchController.searchBar.text!
```

Unwrapping con Optional Binding

Podemos acceder al valor de una variable opcional en una estructura if, y hacer alguna acción en función de si el valor era nulo o no.

```
if let index = meals.index(of: filteredMeals[selectedIndexPath.row]) {  
  
}
```

Funciones como parámetro

Uso de funciones como argumento de otras funciones. Lo utilizaremos para filtrar comidas para llevar a cabo la búsqueda. En nuestro caso será además una función lambda.

Funciones lambda (Closure)

Funciones locales que se pueden utilizar como argumento.

```
filteredMeals = meals.filter({(meal: Meal) -> Bool in
    if scope == "Name" {
        return meal.name.lowercased().contains(searchText.lowercased())
    } else if scope == "Rating" {
        return searchText == String(meal.rating)
    } else {
        fatalError("Received unknown scope: \(scope)")
    }
})
```

Protocolos

Define un modelo mediante propiedades y métodos necesarios para realizar una tarea concreta. Es similar al concepto de interfaz.

```
protocol UISearchResultsUpdating {  
    ...  
    func updateSearchResults(for searchController: UISearchController) {  
    }  
    ...  
}
```

Controlador

Es un objeto que actúa como intermediario entre una o más vistas y uno o más modelos.

Conduce los cambios en el modelo hacia la vista y viceversa. Pueden realizar tareas de configuración y coordinación, así como manejar el ciclo de vida de otros objetos.

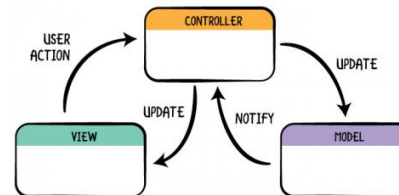


Table view

Lista dinámica de elementos en forma de tabla. Utilizaremos métodos que necesitan la siguiente información:

- » Número de comidas por sección: en función de si estamos filtrando o no, este número variará.
- » Celda en una fila concreta: en función también de si hay una búsqueda activa o no habrá que coger la comida de una fuente u otra. Hay que devolver una celda para poder ser visualizada en la tabla, con todos los atributos que creamos convenientes.

Unwind segue

Es una navegación hacia atrás, de tal forma que en lugar de crear una nueva vista lo que se hace es volver a una vista anterior.

En nuestro caso será necesario para añadir y actualizar comidas, de tal forma que la tabla recibirá una comida añadida o actualizada que se extraerá en el método `unwindToMealList` (es donde llama la vista de añadir y actualizar comidas para hacer efectivo el segue unwind).

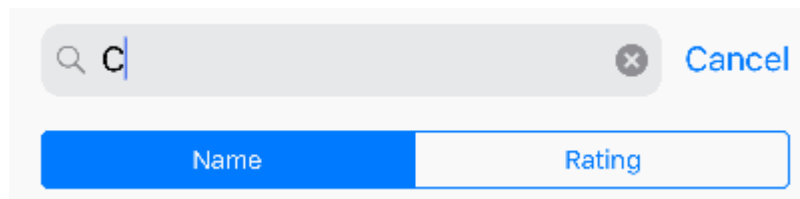
Search bar

Acepta texto como entrada que puede utilizarse en una búsqueda. Nosotros utilizaremos estos componentes:

- » Campo de texto para búsqueda.
- » Botón de limpiado.
- » Título descriptivo.

Scope bar

Define el ámbito de la búsqueda y se combina con la barra de búsqueda. En esta barra aparecen categorías claramente definidas, que en nuestro caso serán 'Name' y 'Rating'.



The image shows a UI mockup of a search interface. At the top is a search bar with a magnifying glass icon, the letter 'C', a clear button (an 'x' in a circle), and a 'Cancel' button. Below the search bar is a scope bar, which is a horizontal container with two segments: a blue segment labeled 'Name' and a white segment labeled 'Rating'.

Implementación de la búsqueda

10

Dado que la barra de búsqueda aparecerá en la tabla de comidas, sólo utilizaremos la clase **MealTableViewController**.

```
class MealTableViewController: UITableViewController, UISearchResultsUpdating {  
    ...  
    var filteredMeals = [Meal]()  
    let searchController = UISearchController(searchResultsController: nil)  
    ...  
    func updateSearchResults(for searchController: UISearchController) {  
    }  
}
```

Heredamos del protocolo **UISearchResultsUpdating** para poder responder a los eventos de actualización de la barra de búsqueda.

El único método que nos pide implementar es **updateSearchResults**, al cual se llamará cuando la barra de búsqueda pase a estar activa o haya cambios en ella.

También tendremos que declarar un controlador de la búsqueda (**UISearchController**), el cual contendrá, entre otras cosas, la barra de búsqueda, así como un nuevo vector, **filteredMeals**, que representa el resultado de la búsqueda actual.

Implementación de la búsqueda

11

En el método **viewDidLoad** habrá que declarar las propiedades que tendrá el espacio de búsqueda.

```
searchController.searchResultsUpdater = self  
  
searchController.observesBackgroundDuringPresentation = false  
  
searchController.searchBar.placeholder = "Search meal"  
  
navigationItem.searchController = searchController  
  
navigationItem.hidesSearchBarWhenScrolling = false  
  
definesPresentationContext = true
```

Clase que reciba información al actualizarse la barra de búsqueda. También se deriva del protocolo **UISearchResultsUpdating**.

Útil cuando las vistas donde se muestran la barra de búsqueda y los resultados son diferentes. No es nuestro caso.

Texto por defecto en la barra de búsqueda.

Especificamos en la navegación nuestro controlador de búsqueda.

La barra de búsqueda aparece permanentemente al marcarlo como false.

Evita que la barra de búsqueda aparezca incluso al cambiar a otras vistas.

Implementación de la búsqueda

12

Algunos métodos de ayuda en la búsqueda:

```
private func searchBarIsEmpty() -> Bool {  
    return searchController.searchBar.text?.isEmpty ?? True  
}
```

Comprueba si la barra de búsqueda está vacía.

```
private func isFiltering() -> Bool {  
    return searchController.isActive && !searchBarIsEmpty()  
}
```

Determina si la búsqueda está activa. No sólo porque el controlador esté activo (barra de búsqueda en primer plano) sino también porque hay un filtro (texto no vacío).

```
private func filterContentForSearchText(_ searchText: String, scope: String = "Name") {  
    filteredMeals = meals.filter({(meal: Meal) -> Bool in return  
        meal.name.lowercased().contains(searchText.lowercased()) })  
    tableView.reloadData()  
}
```

Selecciona aquellas comidas que corresponden con la búsqueda realizada. En nuestro caso consideramos tanto el nombre de la comida como el texto de la búsqueda en minúscula, pero se puede llevar a cabo cualquier otra estrategia.

Implementación de la búsqueda

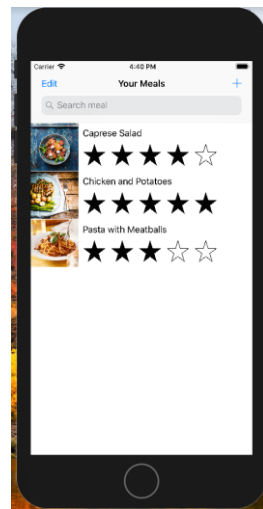
13

¿Dónde se utilizará el filtro? Lo utilizaremos en la función que anteriormente dejamos vacía del protocolo `UISearchResultsUpdating`.

```
func updateSearchResults(for searchController: UISearchController) {  
    filterContentForSearchText(searchController.searchBar.text!)  
}
```

Llegados a este punto tenemos una vista con la barra de búsqueda.

Aunque las comidas se están filtrando y guardando en el vector `filteredMeals`, aún no se muestran los resultados.



Ejemplos de búsqueda:

'Sal' => 'Caprese Salad'

'Cap' => 'Caprese Salad'

'C' => 'Caprese Salad', 'Chicken and potatoes'

Implementación de la búsqueda

14

Para que la **búsqueda funcione** aún nos queda modificar aquellos métodos encargados de montar la tabla de comidas, los cuales ya vienen dados en la implementación base:

```
override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    if isFiltering() {
        return filteredMeals.count
    }
    return meals.count
}

override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    guard let cell = ...

    let meal : Meal
    if isFiltering() {
        meal = filteredMeals[indexPath.row]
    } else {
        meal = meals[indexPath.row]
    }

    cell.nameLabel.text = meal.name
    ...
}
```

Siempre devolvíamos el tamaño del vector **meals**, pero si hay una búsqueda en curso habrá que devolver el tamaño del vector de comidas filtrado, **filteredMeals**.

También habrá que seleccionar la comida que debemos mostrar en la tabla, y para ello también habrá que tener en cuenta si la búsqueda está activa.

Implementación de la búsqueda

15

Aún nos queda actualizar el borrado de filas...

```
override func tableView(_ tableView: UITableView, committedEditingStyle: UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {  
    if editingStyle == .delete {  
        if isFiltering() {  
            if let index = meals.index(of: filteredMeals[indexPath.row]) {  
                meals.remove(at: index)  
            }  
  
            filteredMeals.remove(at: indexPath.row)  
        } else {  
            meals.remove(at: indexPath.row)  
        }  
  
        saveMeals()  
        tableView.deleteRows(at: [indexPath], with: .fade)  
    }  
}
```

Habrán dos situaciones claras:

- **Búsqueda activa:** conocemos el índice de la comida a eliminar en `filteredMeals`, no en `meals`. Aún así, podemos extraer qué índice es el de la comida seleccionada en `meals`.
- **Búsqueda inactiva:** sólo actualizamos el vector principal, `meals`.

Implementación de la búsqueda

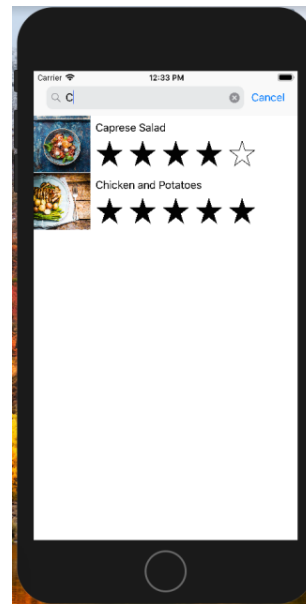
16

Nuestra búsqueda ya funciona



Ya podemos hacer búsquedas, pero aún **nos queda la comunicación con otras vistas**. En concreto, a la vista determinada por `MealViewController` habrá que pasarle la comida correcta si se selecciona una fila para modificarla, y a la hora de volver a nuestra vista, habrá que tener en cuenta que la comida actualizada puede no encajar en la búsqueda en curso (si hay una).

Nosotros consideraremos que se puede realizar la acción de actualizar y eliminar mientras haya una búsqueda en curso.



Implementación de la búsqueda

17

El primero de los métodos que participa en la navegación con MealViewController es **prepare**, el cual pasará la comida correspondiente en caso de que la acción sea la de mostrar los detalles de una comida.

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    super.prepare(for: segue, sender: sender)

    switch(segue.identifier ?? "") {
    case "AddItem":
        os_log("Adding a new meal.", log: OSLog.default, type: .debug)
    case "ShowDetail":
        ...
        let selectedMeal: Meal
        if isFiltering() {
            selectedMeal = filteredMeals[indexPath.row]
        } else {
            selectedMeal = meals[indexPath.row]
        }
        mealDetailViewController.meal = selectedMeal
    default:
        fatalError("Unexpected Segue Identifier; \(String(describing: segue.identifier))")
    }
}
```

De nuevo, se trata sólo de elegir una comida de un vector u otro dependiendo de la situación de la búsqueda.

Implementación de la búsqueda

18

El segundo método será `unwindToMealList`, donde se recibirá una comida, actualizada o nueva.

Actualización

```
var removeRow = false

if isFiltering() {
    if let index = meals.index(of: filteredMeals[selectedIndexPath.row]) {
        meals[index] = meal

        if !mealMatchesSearch(meal: meal, searchText: searchController.searchBar.text!) {
            filteredMeals.remove(at: selectedIndexPath.row)
            removeRow = true
        }
    }
} else {
    meals[selectedIndexPath.row] = meal
}

if !removeRow {
    tableView.reloadRows(at: [selectedIndexPath], with: .none)
}
```

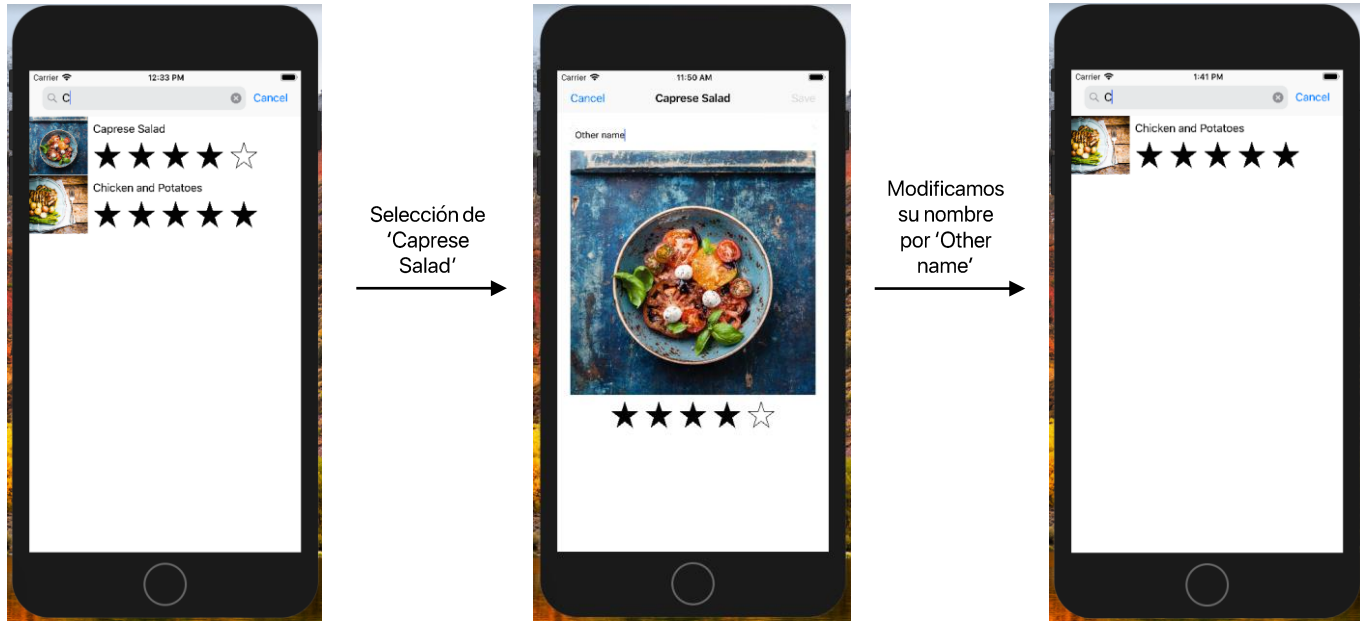
Es posible que la comida actualizada no encaje en la búsqueda activa, y esto es lo que se comprueba en `mealMatchesSearch`. Si no encaja, se elimina y no es necesario actualizar esa fila de la tabla.

Inserción

```
let newIndexPath: IndexPath?

meals.append(meal)
tableView.insertRows(at: [IndexPath], with: .automatic)
```

Al contrario que la actualización, no se puede añadir mientras haya una búsqueda activa, por lo que siempre insertaremos en `meals`. `newIndexPath` especifica la fila donde aparecerá la nueva comida en la tabla.



La búsqueda está completa, y una posible escena es ésta.

Búsqueda por campos

20

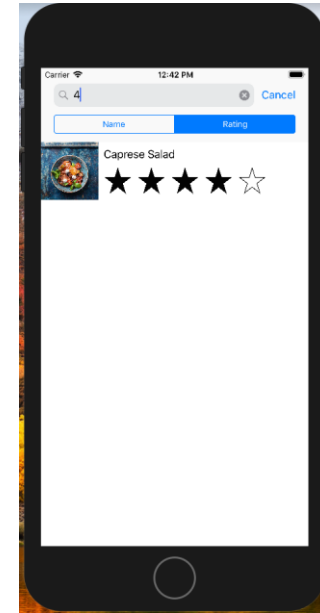
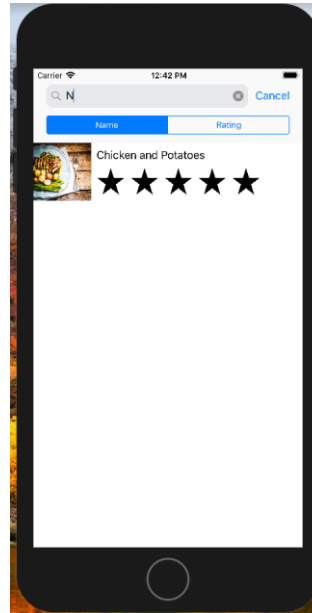
Se puede filtrar aún más la búsqueda...



En nuestra aplicación puede resultar un poco más difícil incorporar esta funcionalidad, pero un buen ejemplo sería un FoodTracker donde cada comida tuviera una categoría: Entrante, Postre...

Podríamos entonces buscar por categorías específicas en lugar de en toda la lista. Se podría dejar además una opción de buscar en todo el conjunto, tal y como funciona nuestra aplicación ahora mismo.

En nuestro caso, los campos que hemos usado son Nombre y Valoración.



Búsqueda por campos

21

Las **modificaciones** que debemos realizar para incorporar esta funcionalidad son:

Método viewDidLoad

```
searchController.searchBar.scopeButtonTitles = ["Name", "Rating"]
searchController.searchBar.delegate = self
```

Método que necesitamos implementar debido al protocolo UISearchResultsUpdating

```
func updateSearchResults(for searchController: UISearchController) {
    let searchBar = searchController.searchBar
    let scope = searchBar.scopeButtonTitles![searchBar.selectedScopeButtonIndex]
    filterContentForSearchText(searchController.searchBar.text!, scope: scope)
}
```

Método para elegir el resultado de una búsqueda. Ahora tendremos que diferenciar entre los dos campos

Importante: en 'Rating' se transforma a String la valoración de la comida, y no a Int la cadena de búsqueda, ya que podría fallar en la conversión.

```
private func filterContentForSearchText(_ searchText: String, scope: String = "Name") {
    filteredMeals = meals.filter({(meal: Meal) -> Bool in
        if scope == "Name" {
            return meal.name.lowercased().contains(searchText.lowercased())
        } else if scope == "Rating" {
            return searchText == String(meal.rating)
        } else { return false }})
    tableView.reloadData()
}
```

Las **modificaciones** que debemos realizar para incorporar esta funcionalidad son:

Al marcar nuestra clase `MealTableViewController` como **delegate** de la barra de búsqueda, habrá que implementar el protocolo `UISearchBarDelegate`.

```
class MealTableViewController: UITableViewController, UISearchResultsUpdating, UISearchBarDelegate {  
    ...  
    func searchBar(_ searchBar: UISearchBar, selectedScopeButtonIndexDidChange selectedScope: Int) {  
        filterContentForSearchText(searchBar.text!, scope: searchBar.scopeButtonTitles![selectedScope])  
    }  
}
```

Método para saber si una comida encaja en una búsqueda concreta

```
private func mealMatchesSearch(meal: Meal, searchText: String) -> Bool {  
    let scope = searchController.searchBar.scopeButtonTitles![searchController.searchBar.selectedScopeButtonIndex]  
    if scope == "Name" {  
        return meal.name.lowercased().contains(searchText.lowercased())  
    } else if scope == "Rating" {  
        return String(meal.rating).contains(searchText)  
    } else {  
        fatalError("Receieved unknown scope: \(scope)")  
    }  
}
```



PROYECTO EN GITHUB

[iOS_SearchBar](#), [AlfonsoLRz](#)