Swift

REALIZACIÓN DE BÚSQUEDAS Y FILTRADOS

Alfonso López Ruiz
alr00048@red.ujaen.es

En este minitutorial se pretende englobar los pasos necesarios para añadir una barra de búsqueda en la aplicación base de FoodTracker. Dicha barra nos deberá permitir filtrar todas las comidas guardadas en función de su nombre, de tal forma que sólo se muestren aquellas que coinciden con el texto introducido. Adicionalmente, también se podrá elegir si filtrar por nombre o por puntuación.

FUNDAMENTOS TEÓRICOS

Los fundamentos teóricos necesarios para la correcta comprensión de este minitutorial son los siguientes:

- **Swift**:
  - **Constantes (let):** variables inmutables.
  - **Arrays**: conjunto lineal de elementos accesibles a través de índices.
    Ejemplo: meal = ["Caprese Salad", "Chicken and Potatoes"]

    Utilizaremos especialmente los métodos **remove** (actualizaciones y borrados) e **index**(of:) para identificar el índice de una comida del vector filtrado de comidas en el vector principal. También utilizaremos el atributo **count** para conocer el tamaño de un vector.
  - **Funciones**: bloques de código con un propósito muy concreto. Nos interesa especialmente conocer que estas funciones pueden declaradas como privadas con la palabra **private** antes de la función.

    También nos interesan los **valores por defecto de los argumentos** de la función.
  - **Variables opcionales (Optionals):** variables que pueden contener un valor o no (nil). Ejemplo: var newIndexPath: IndexPath?
  - **Obtener el valor de una variable opcional (Unwrapping):** sea newIndexPath una variable opcional, accederemos a su valor con el carácter ! Ejemplo: newIndexPath!
  - **Unwrapping mediante Optional Binding (en una estructura if):** de esta forma podemos acceder al interior de una estructura if si la variable tiene un valor distinto de nulo. Ejemplo:

    ```
    if let index = newIndexPath { /* TO DO, index no nulo. */ }
    ```
  - **Funciones como parámetro de otras funciones**: lo utilizaremos para filtrar la comida después de una búsqueda. Nosotros además especificaremos una función lambda.
  - **Funciones lambda (Closure):** nos permite crear funciones locales que se pueden utilizar como argumentos. La sintaxis es la siguiente:

    ```
    { ( parameters ) -> return type in
        statements
    }
    ```
- **Bars**:
  - **Search bar**: acepta texto como entrada que puede utilizarse en una búsqueda. Nosotros utilizaremos estos componentes:
    - **Campo de texto para búsqueda**.

- **Botón para 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 Nombre y Valoración.

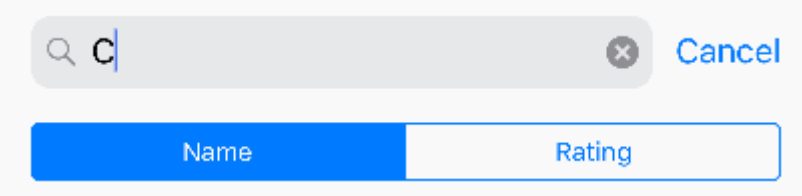
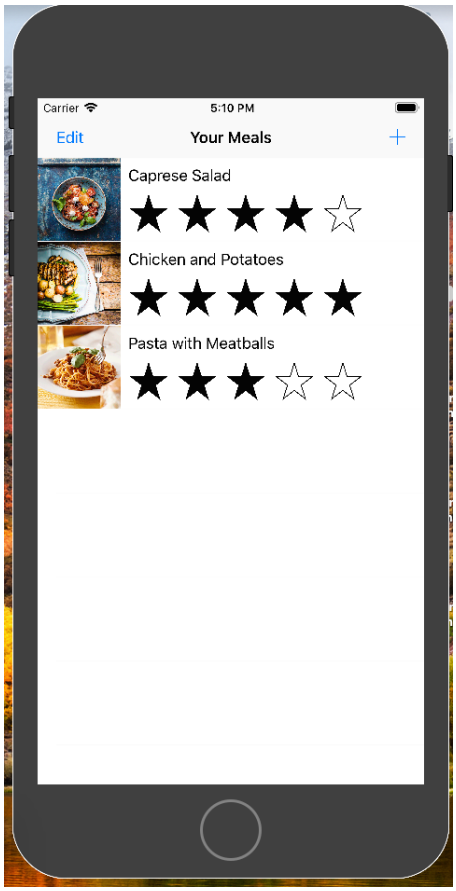


Figura 1. La barra de búsqueda y los campos que implementaremos.

IMPLEMENTACIÓN DE LA BARRA DE BÚSQUEDA

El punto de partida para nuestra aplicación será, como ya sabemos, la aplicación FoodTracker de las que nos provee Apple (https://developer.apple.com/sample-code/swift/downloads/09_PersistData.zip). En este momento la pantalla de nuestra aplicación debería quedar como sigue:



La pantalla que tendremos que modificar para incluir nuestra barra de búsqueda será la correspondiente a la tabla de comidas, la cual es gestionada por el archivo **MealTableViewController.swift**.

El elemento que vamos a introducir forma parte también de la navegación que ya está incluida, por lo que nuestro primero paso será modificar dicha parte de la vista en el método viewDidLoad de la clase ya mencionada.

Figura 2. Apariencia de la aplicación base.

Nuestro primer paso será incluir el controlador de la barra de búsqueda (**UISearchController**) como un atributo en nuestro archivo MealTableViewController:

```
let searchController = UISearchController(searchResultsController: nil)
```

Como argumento recibirá la vista que se encarga de mostrar el resultado de la búsqueda. En nuestro caso, la vista donde se busca y donde se muestran los resultado es la misma, por lo que utilizaremos *nil*.

También necesitaremos que nuestra clase implemente el protocolo **UISearchResultsUpdating** para poder responder a los eventos que suceden en la barra de búsqueda. El único método que tendremos que implementar es el siguiente:

```
class MealTableViewController: UITableViewController, UISearchResultsUpdating {

    func updateSearchResults(for searchController: UISearchController) {

    }

}
```

Será al iniciarse nuestra tabla de comidas (**viewDidLoad**) donde tengamos que incorporar el controlador de la barra de búsqueda, para lo cual tendremos que utilizar el siguiente bloque de código:

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

- **SearchResultsUpdater**: indica la clase que recibirá información cuando se modifique el contenido de la barra de búsqueda. Esta propiedad se deriva del protocolo **UISearchResultsUpdating**.
- **ObscuresBackgroundDuringPresentation**: es útil en el caso de que las vistas donde se busca y donde se muestran los resultados son diferentes. En nuestro ejemplo, esto no es así y por lo tanto usaremos *false*.
- **HidesNavigationBarDuringPresentation**: por defecto será *true*, y de ser así, al buscar (utilizar la barra de búsqueda) desaparecería la barra de navegación (título, botón de editar y de añadir). En una aplicación donde la barra de navegación sólo mostrara un título podría ser interesante ponerla a verdadera para tener más espacio. En nuestro caso, permitiremos que se editen comidas que aparecen como resultado de una búsqueda, y también añadir, por lo que no podemos ocultar esta barra.
- **Placeholder**: será el texto que se muestre en la barra de búsqueda cuando esté vacía. Hemos elegido la cadena 'Search meal' pero también podríamos elegir una cadena a modo de ejemplo de lo que podemos buscar: 'Caprese Salad'.
- **Search controller**: como ya comentamos, la barra de búsqueda formará parte de la navegación, por lo que habrá que especificar en `navigationItem` cuál es nuestro controlador de búsqueda.

- **HidesSearchBarWhenScrolling**: lo pondremos a false ya que de no ser así la barra de búsqueda no saldría desde un comienzo. Es una forma de dejarla permanente en la vista.
- **DefinesPresentationContext**: evita que la barra de búsqueda aparezca incluso cuando cambiamos a otras vistas.

Como ya sabemos, guardamos un vector de comidas que representa la totalidad de las que disponemos. Sin embargo, el resultado de la búsqueda será un subconjunto de éstas, por lo que será necesario otro vector que guarde dicho subconjunto: filteredMeals.

Comenzamos a implementar métodos que nos servirán de ayuda en la búsqueda:

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

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

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

Es de especial importancia el segundo método, el cual podemos personalizar como deseemos. Actualiza el vector de resultados de la búsqueda y determina mediante una función definida inline qué elementos y cuales no forman parte del vector de resultados.

Será una función que devuelve un booleano, y en nuestro caso transformamos tanto el nombre de la comida como el texto de la barra a minúsculas. Devolveremos true si el nombre de la comida contiene el texto introducido por el usuario.
Por ejemplo, 'Cap' o 'Sal' devolvería 'Caprese Salad'.

El argumento scope determinará, por ejemplo, el campo por el que estamos buscando: Nombre, valoración... Para la implementación básica no es necesario conocer para qué es esta variable, pero sí que aparecerá en el último apartado opcional.

Por último, necesitamos un método para determinar cuándo está activa la búsqueda, ya que nos ayudará a la hora de saber qué vector utilizar, meals o filteredMeals. Lo veremos más tarde.

¿Dónde usaremos este filtro? Se usará en la función updateSearchResults que antes dejamos vacía.

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

Podemos hacer una pausa y ver como quedaría nuestra aplicación en este momento:

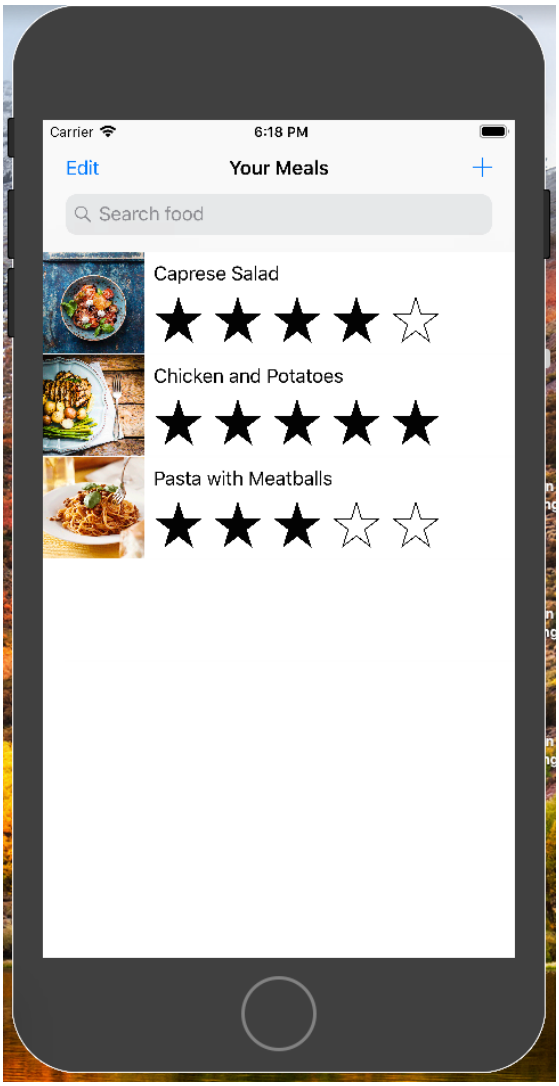


Figura 3. Apariencia de nuestra aplicación con la barra de búsqueda.

Llegados a este paso tenemos nuestra barra de búsqueda pero ésta no es funcional ya que sólo estamos utilizando los datos del vector meals, aunque sí se está actualizando el vector filteredMeals.

Primero gestionaremos cómo mostrar los resultados, y después necesitaremos modificar la comunicación con otras vistas.

Modificaremos los siguientes métodos ya implementados en el proyecto base, como sigue:

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

    return meals.count
}
```

Primero tenemos que controlar qué número de filas se van a mostrar. Dependiendo de si la búsqueda está activa se extraerá este número de un vector u otro.

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cellIdentifier = "MealTableViewCell"

    guard let cell = tableView.dequeueReusableCell(withIdentifier: cellIdentifier, for: indexPath) as?
MealTableViewCell  else {
        fatalError("The dequeued cell is not an instance of MealTableViewCell.")
    }

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

    cell.nameLabel.text = meal.name
    cell.photoImageView.image = meal.photo
    cell.ratingControl.rating = meal.rating

    return cell
}
```

Cada una de las filas adoptará la información de una comida, por lo que tendremos que saber de dónde extraer (de qué vector) las comidas. De nuevo, es la misma situación: de dónde extraer la información.

Por último nos queda la eliminación de filas:

```
override func tableView(_ tableView: UITableView, commit editingStyle: 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)
    } else if editingStyle == .insert {
      // Create a new instance of the appropriate class, insert it into the array, and add a new row to the table view
    }
}
```

Habrá que distinguir dos situaciones muy diferentes:

- **La búsqueda está activa**, luego el índice que nos devuelve el evento es del vector filteredMeals, por lo que para actualizar el vector meals habrá que hacer una operación un poco más compleja que en el siguiente caso.
- **La búsqueda no está activa**, se actualiza el vector principal meals.

Llegados a este punto deberíamos ser capaces de buscar y filtrar nuestras comidas, como en la siguiente pantalla:

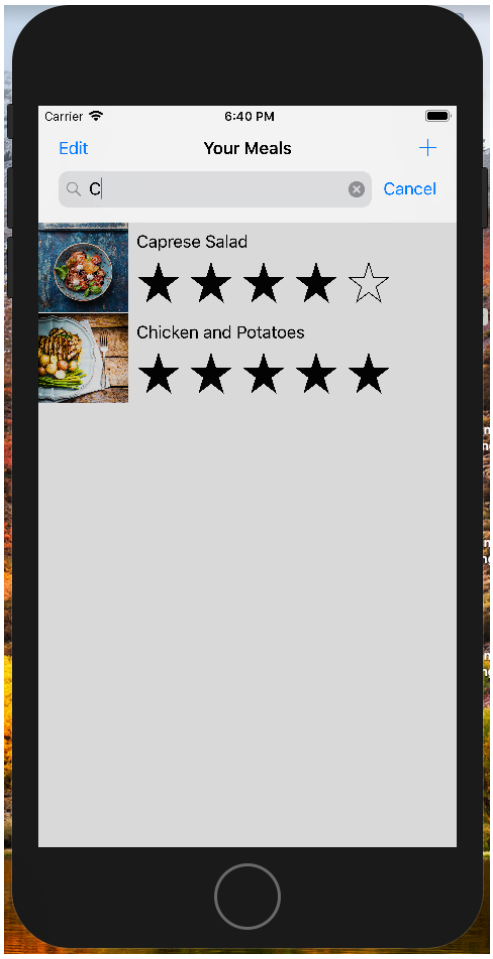


Figura 4. Filtrado de comidas en nuestra aplicación.

Nuestro último paso será adoptar un comportamiento correcto para comunicarnos con las vistas de añadir y actualizar.

Muy importante. Nuestro enfoque es el siguiente: permitiremos añadir y actualizar comidas una vez se han mostrado resultados de la búsqueda, teniendo en cuenta que modificar o añadir nuevas comidas puede hacer que se modifique la lista de resultados obtenidos.

Otro enfoque: no permitir añadir pero sí editar (que puede ser más normal). No lo hemos implementado pero como vistazo rápido a este enfoque podemos decir que se pueden habilitar y deshabilitar los botones de la barra de navegación:

navigationItem.rightBarButtonItem.isEnabled = true / false

Además, podemos deshabilitar el botón cuando se llame al método updateSearchResults, al cual se acude siempre que haya cambios en la barra, incluyendo los eventos de limpiar y cancelar. Por lo que si el texto de la barra está vacía podemos habilitar de nuevo el botón.

Primero tendremos que modificar el método prepare ya que necesitamos distinguir la comida que le pasaremos a **MealViewController**. De nuevo, si la búsqueda está activa escogeremos la comida del vector filteredMeals, si no lo haremos de meals. Nos debería quedar así:

```
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":
      guard let mealDetailViewController = segue.destination as? MealViewController else {
        fatalError("Unexpected destination: \(segue.destination)")
      }

      guard let selectedMealCell = sender as? MealTableViewCell else {
        fatalError("Unexpected sender: \(String(describing: sender))")
      }

      guard let indexPath = tableView.indexPath(for: selectedMealCell) else {
        fatalError("The selected cell is not being displayed by the table")
      }

      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))")
    }
  }
```

Llegados a este punto sólo nos queda la actualización de las comidas que tenemos, tanto si se añade como si actualiza alguna de las que ya guardamos. Como ya comentamos, es posible que si hay una búsqueda activa, los resultados de la misma varíen (+1 si se añade una comida que encaje con la búsqueda y -1 si se actualiza una comida y deja de encajar con la búsqueda).

Para ello es importante saber si una comida encaja con la búsqueda actual, y para ello implementaremos el siguiente método:

```
private func mealMatchesSearch(meal: Meal, searchText: String) -> Bool {
        return meal.name.lowercased().contains(searchText.lowercased())
    }
```

Sólo nos quedaría modificar el método unwindToMealList, de tal forma que podamos actualizar nuestros vectores de comida:

```
@IBAction func unwindToMealList(sender: UIStoryboardSegue) {
    if let sourceViewController = sender.source as? MealViewController, let meal = sourceViewController.meal {

        if let selectedIndexPath = tableView.indexPathForSelectedRow {
            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)
          }
        }
        else {
          var newIndexPath : IndexPath?
          if isFiltering() && mealMatchesSearch(meal: meal, searchText: searchController.searchBar.text!) {
            newIndexPath = IndexPath(row: filteredMeals.count, section: 0)
            filteredMeals.append(meal)
          } else if !isFiltering(){
            newIndexPath = IndexPath(row: meals.count, section: 0)
          }

          meals.append(meal)
          if let indexPath = newIndexPath {
            tableView.insertRows(at: [indexPath], with: .automatic)
          }
        }

        saveMeals()
      }
    }
```

Habrá que distinguir dos operaciones:

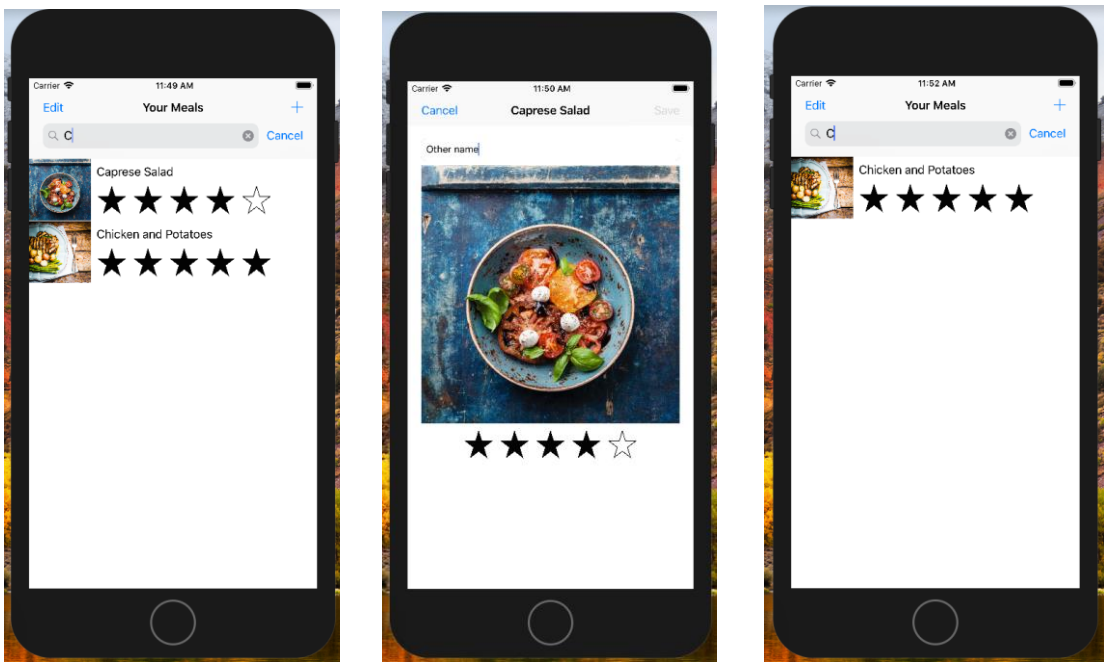
- **Actualización**: habrá que modificar la comida en el vector original `meals` siempre, pero si hay una búsqueda activa es posible que tengamos que eliminar dicha comida del vector filtrado si los nuevos datos no encajan con el texto de la búsqueda.

  Parece obvio que si se elimina una comida del vector que sea no habrá que actualizar dicha fila en la tabla, y esto se controla con la variable `removeRow`.

- **Añadir una nueva comida**. Nuevamente, siempre habrá que añadir una comida al vector principal `meals`, pero sólo se actualizará `filteredMeals` si encaja con la búsqueda activa.

  `IndexPath` marcará el índice donde se añadirá dicha comida.

Llegados a este punto nuestra aplicación debería responder como sigue:



1 2 3

Figura 5. Escena de actualización de una comida.

FILTRAR POR DIFERENTES CATEGORÍAS (EXTRA)

En ciertas aplicaciones puede ser necesario buscar por diferentes campos o categorías. Por ejemplo, si nuestras comidas tuvieran categorías podríamos dividir la búsqueda en Entrantes, Postres... En nuestra aplicación no existen dichas categorías pero para demostrar como se implementa esta funcionalidad podremos buscar por nombre y por valoración.

Nuestro primer paso será modificar el método viewDidLoad, donde habrá que añadir las diferentes categorías de las que dispondremos.

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

También tendremos que modificar algunos de los métodos ya implementados. Comenzaremos por updateSearchResults, del protocolo **UISearchResultsUpdating**:

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

Anteriormente, al método filterContentForSearch lo llamábamos sin argumento scope, ya que tenía un valor por defecto. De todos modos, ni lo utilizábamos dentro de ese método. Ahora esto cambia y nos queda así:

```
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 {
            fatalError("Received unknown scope: \(scope)")
          }
        })

        tableView.reloadData()
    }
```

Nuestra función inline se hace más grande para comprobar la categoría seleccionada. Importante: a la hora de comprobar la valoración, no se transforma el texto introducido por el usuario a entero (que podría fallar), sino la valoración de la comida a String.

Por último, también teníamos un método para comprobar si la nueva comida (añadida o actualizada) encajaba con la búsqueda activa (si la hay). Habrá que tener ahora en cuenta las dos categorías disponibles:

```
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("Recevied unknown scope: \(scope)")
    }
}
```

En este punto nuestra aplicación nos quedaría asi:

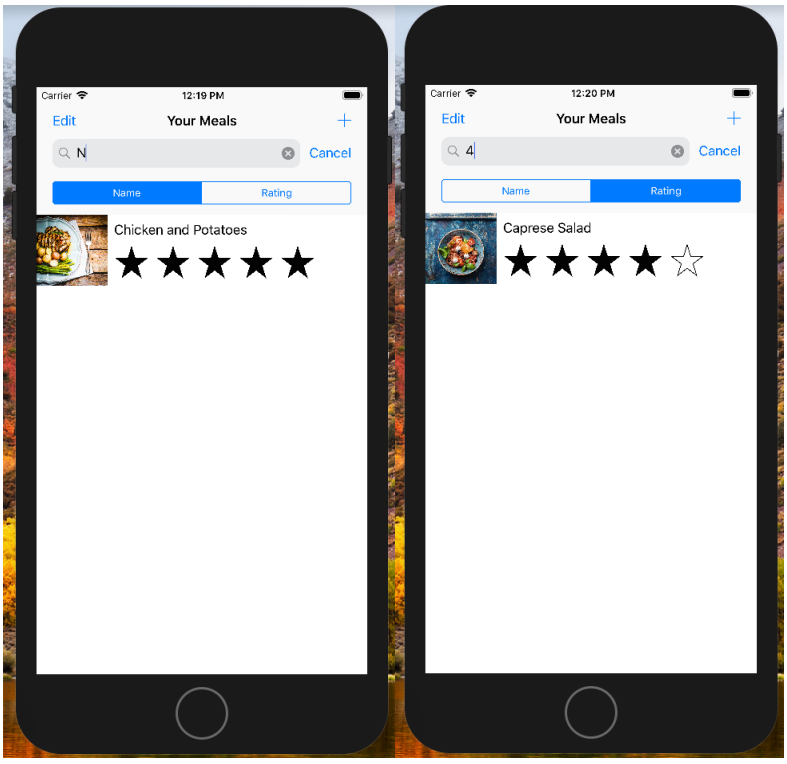


Figura 6. Apariencia de nuestra aplicación una vez implementada la búsqueda y el filtro por categorías.