Le Baluchon Bonus

Concept

Le Baluchon est une application visant à faciliter le voyage de l'utilisateur en lui fournissant des informations météo, un convertisseur de taux de change et un traducteur automatique

Bonus

Les bonus de l'application sont les suivants :

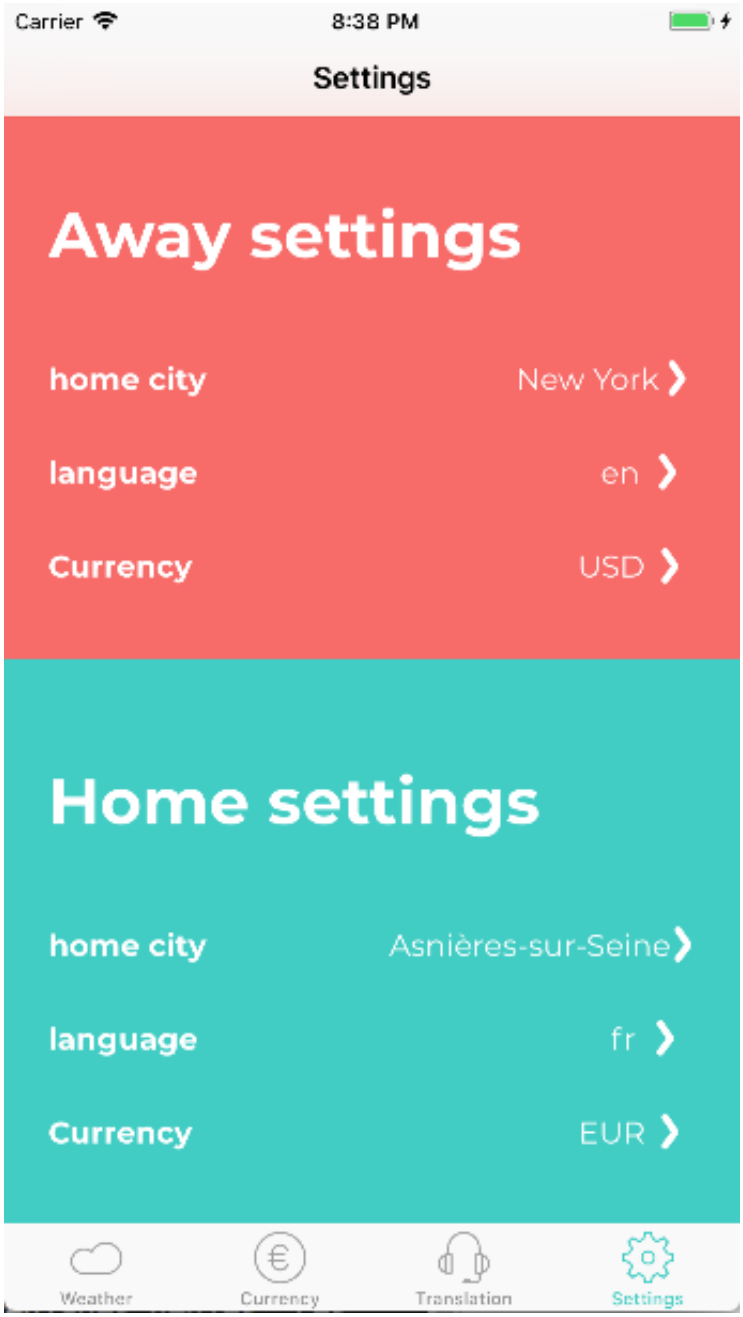
1. Une page de reglage permettant de choisir devise, langue, et localisation pour le pays hôte et le pays de départ
2. Un Convertisseur de devise
3. L'auto-detection du langage de l'utilisateur à la saisie du texte à traduire

User Settings

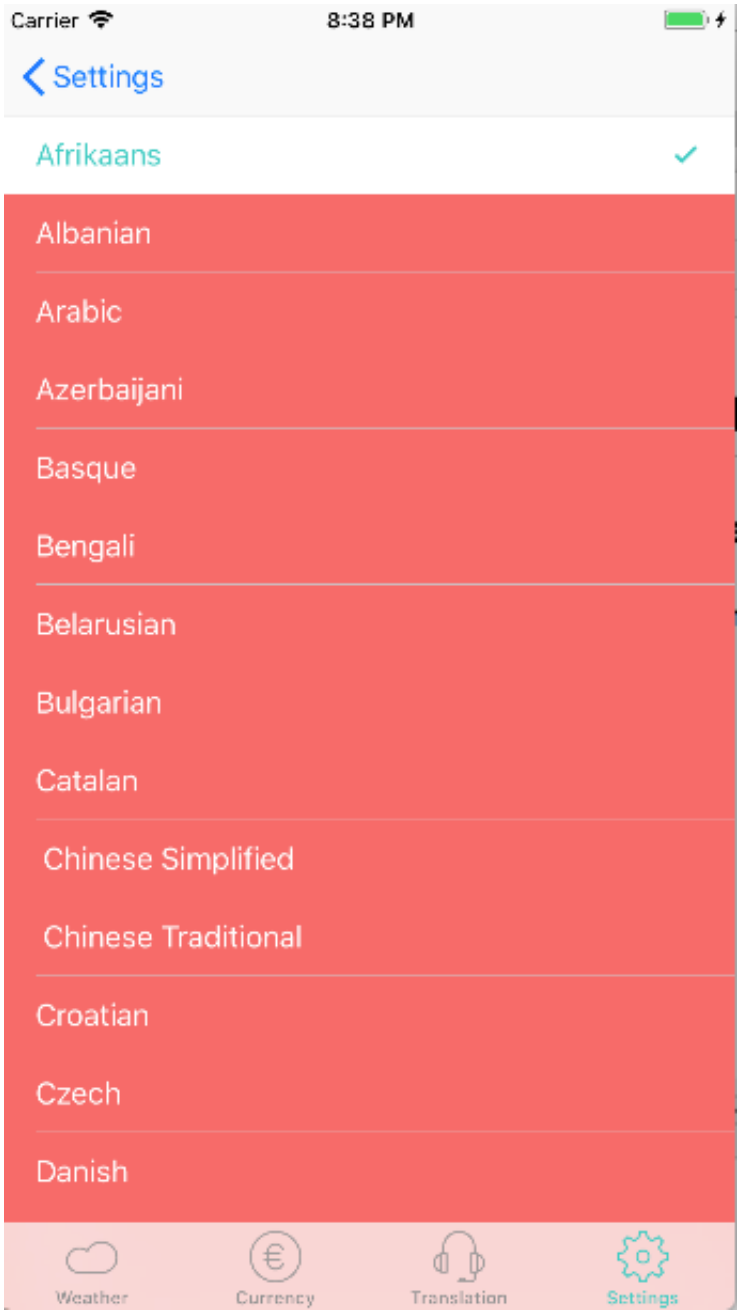
Creation des MVC

Nous créons une la structure MVC suivante pour le user settings


UserSettingsController
View: UserSettingsController.xib
Model: UserSettings.swift
LocationSettingsrController
CurrencySettingsController
LangugaSettingsController
View: WeatherSettingsController.xib
MODEL
Fetching Data LocationService.swift
Data Model LocatioModel.swift
View: CurrencySettingsController.xib
Model: CurrencyList.swift
View: LangugafeSettingsController.xib
Model: Language.swift

L'acceuil des user settings est présenté ainsi 

Chaque paramètre donne accès à une page de reglages dédiés. Les possibilités sont présentées dans une tableView.



L'etat de "highlight" de la cellule a été customisé. Un background blanc, une couleur complémentaire et un checkMark pour signalé la sélection.

```
  func highlight(_ cell: UITableViewCell, with color: UIColor){
    // Set the higlight color scheme
    cell.tintColor = color
    cell.textLabel?.textColor =  color
    // Set cell background to white
    cell.backgroundColor = .white
    let bgColorView = UIView()
    bgColorView.backgroundColor = UIColor.white
    cell.selectedBackgroundView = bgColorView
    // add a check mark
    cell.accessoryType = .checkmark
  }
```

Selon l'univers "home" ou "away", le schema est coloriel est différent. Pour cela un initialiseur complémentaire est ajouté dans les view Controller de settings

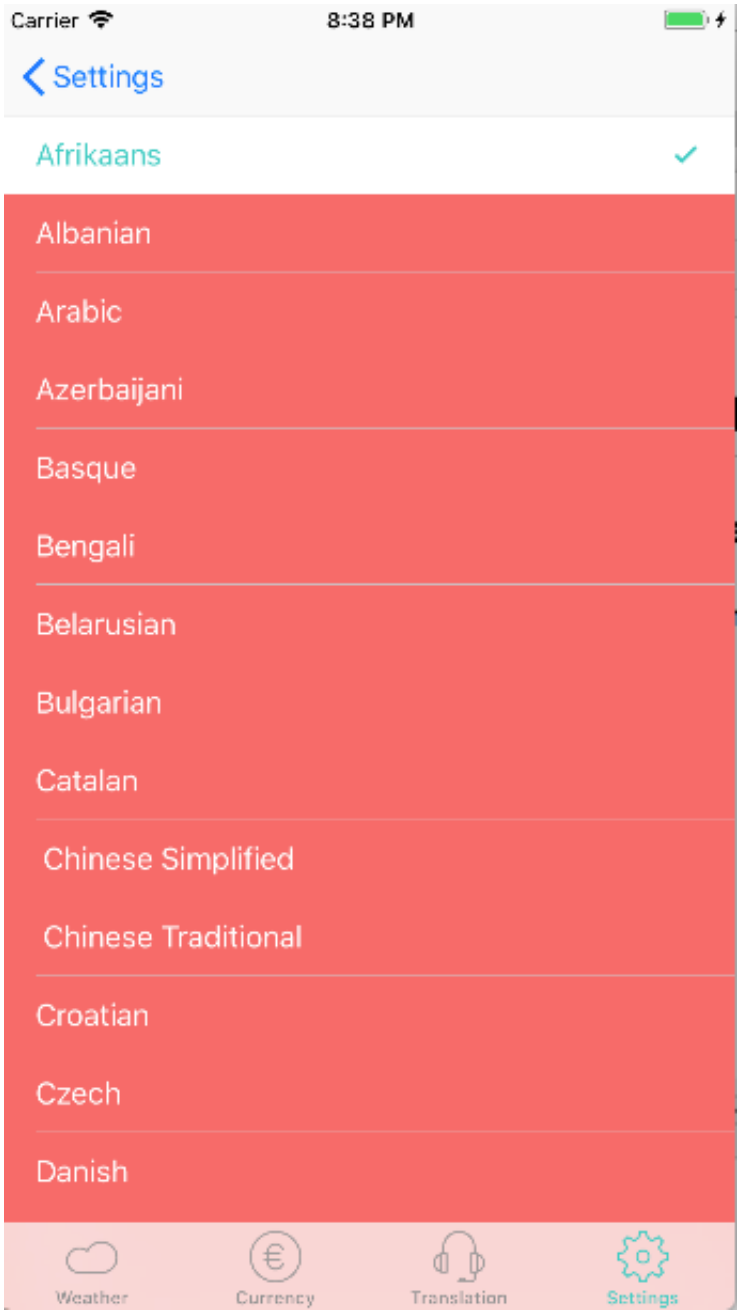
```
  func initDetail(bgColor:UIColor, txtSelect: UIColor, destination: String){
    self.backgroundColor = bgColor
    self.selectedTextColor = txtSelect
    self.source = destination
  }
```

Le schema coloriel est lui initialisé dans le general user Settings au moment ou l'utilisateur sélectionne une propriété à modifier. Ensuite le ViewController push les settings ViewController

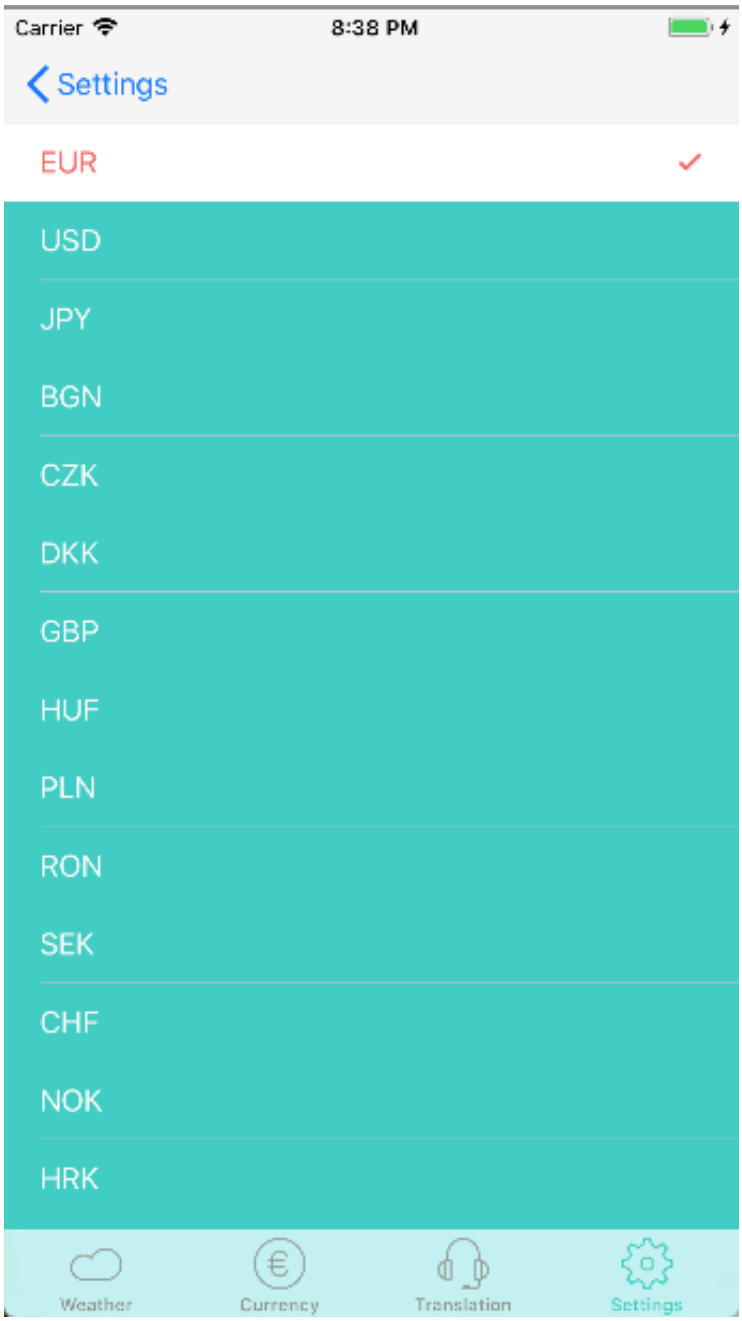
```
  @IBAction func goToCurrencySettings(_ sender: UIButton) {
    currencyListVc.initDetail(bgColor: #colorLiteral(red: 1, green: 0.4196078431,
blue: 0.4078431373, alpha: 1), txtSelect: #colorLiteral(red: 0.2588235294, green:
0.8039215686, blue: 0.768627451, alpha: 1), destination: "away")
    navigationController?.pushViewController(currencyListVc, animated: true)
  }
```

Présentation des schémas coloriels

AWAY


Carrier
8:38 PM
Settings
Afrikaans
Albanian
Arabic
Azerbaijani
Basque
Bengali
Belarusian
Bulgarian
Catalan
Chinese Simplified
Chinese Traditional
Croatian
Czech
Danish
Weather
Currency
Translation
Settings

HOME



Stockage des préférences

Les préférences sont stockées dans le user défault du téléphone.

```
static let defaults = UserDefaults.standard
```

Pour cela un User Settings manager est crée. Il permet de:

1. sauver des options
2. charger des options
3. reinitialiser les options

Au moment ou l'application se lance, les préférences son rechargées.

Afin de bien signaler à l'utilisateur quelles sont les options choisies, quand il entre dans un settingsController son choix est mis en surbrillance. Pour cela nous sauvons l'Integer de l'indexPath.row au moment du choix et le rappelons au moment du viewWillAppear des tableView.

```
 if source == "home" { // home
      selectedHome = UserSettings.loadData(displayKey: homeDisplayKey, indexKey: h
omeIndexKey).1
      home = dataSet[selectedHome!][value]!
      selectRow(selectedHome!)
    } else { // away
      selectedAway = UserSettings.loadData(displayKey: awayDisplayKey, indexKey: a
wayIndexKey).1
      away = dataSet[selectedAway!][value]!
      selectRow(selectedAway!)
    }
```

Localization Settings

Les models pour le languageSettings et le CurrencySettings sont des structs figés en local. Cependant afin d'offrir une meilleur expérience dans le location settings nous fournissons une interface de recherche qui fait appel à une API extérieure : openStreetMap

documentation de l'API openStreetMap

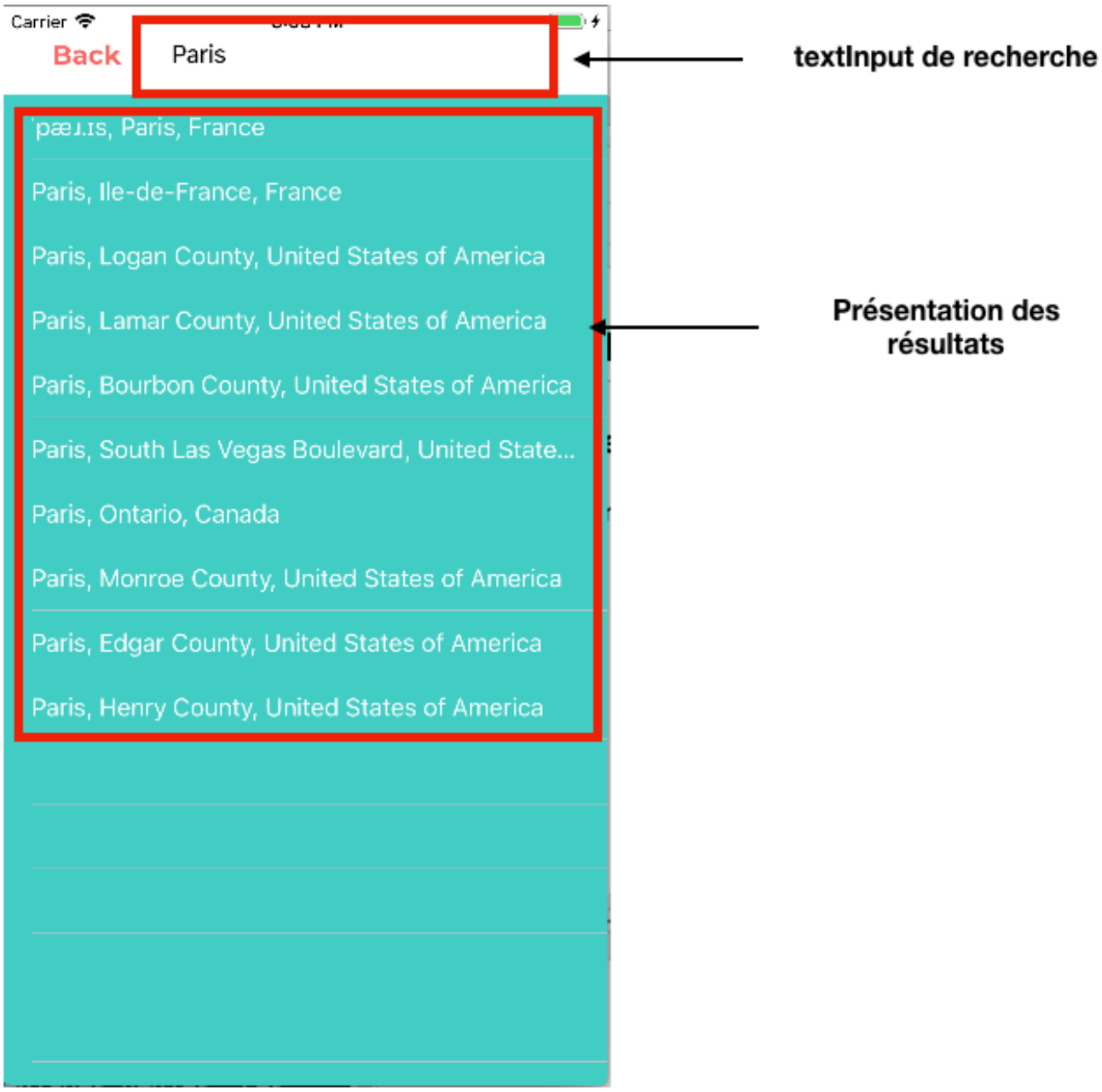
Afin de réaliser les REST API CALL nous utilisons le framework Alamofire.

Repo Github Alamofire & Documentation

L'installation s'est faite via cocoaPods

Tous les appels aux API REST sont fait sur la thread en background. En pratique l'utilisateur, la recherche s'effectue ainsi :

1. L'utilisateur entre le nom d'une ville.
2. il appuie sur la touche entrée.
3. Toutes les propositions sont présentées.
4. Il sélectionne sa ville (elle est alors stockées dans le user defaults)



La validation par la touche entrée ou une touch en dehors du clavier se fait via le UITextField Delegate.

Nous overridons 2 fonctions :

```
  override func touchesBegan(_ touches: Set, with event: UIEvent?) {
    searchLocation()
    table?.isHidden = false
    input?.resignFirstResponder()
     self.view.endEditing(true)
  }



func textFieldShouldReturn(_ textField: UITextField) -> Bool {
   searchLocation()
    table?.isHidden = false
      self.input?.resignFirstResponder()
    return (true)
  }
```

La fonction searchLocation() effectue la requete vers L'API via Alamofire.

Le convertisseur de devise

Le convertisseur est en 3 parties :

1. Un clavier permettant d'effectuer les opérations en partie basse
2. Un affichage des valeurs au milieu
3. Un affichage du taux de change en vigueur et la date du jour

Le model du convertisseur est conenu dans le fichier CurrencyConverter.swift. Ce fichier héberge une structure qui gère toutes les opérations et la logique. Nous faisons le choix d'une structure pour des raisons de mémoires et nous choisissons donc un objet par valeur.

La gestion des boutons est via une collection d'outlet qui renvoie le contenu du *button.titleLable.text.*

Puis cette valeur est stocké dans un table de [String] nommé stringNumbers

Quand l'utilisateur appuie sur la touche convert, il fait appel à la fonction convert() du model

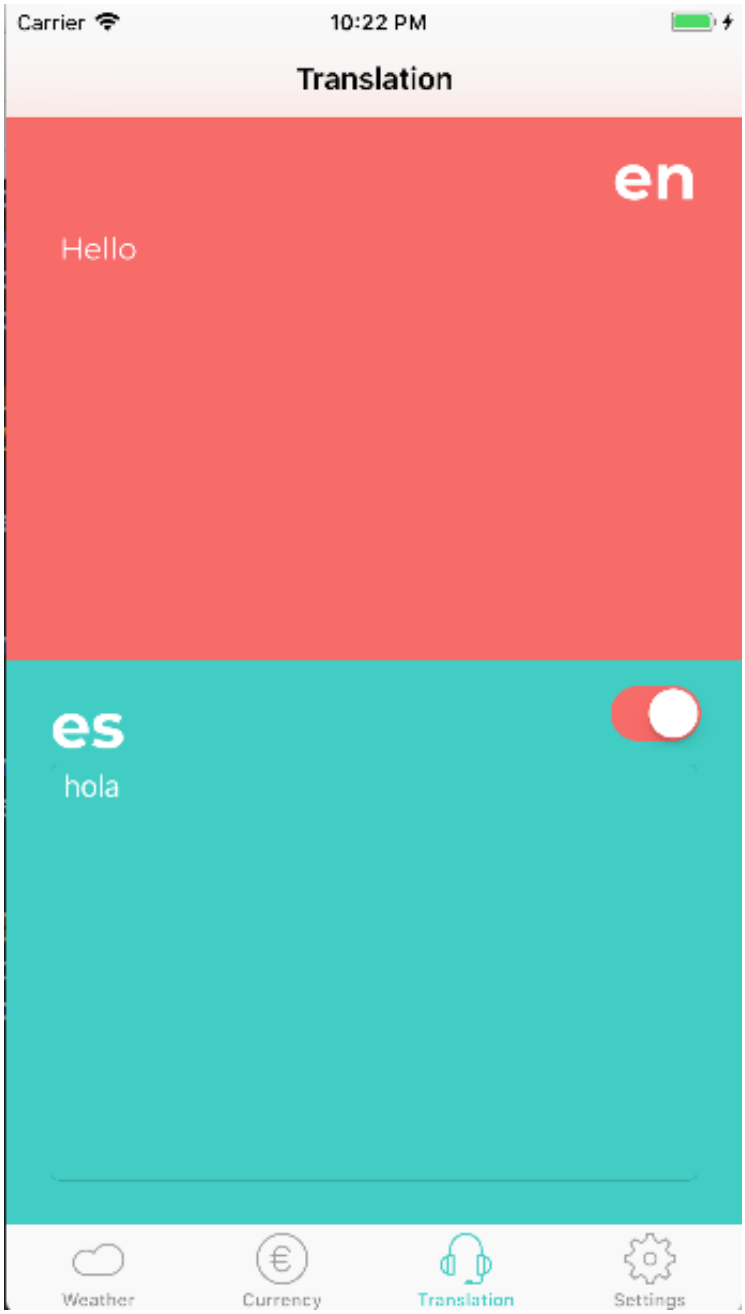
La fonction de conversion est la suivante :

```
mutating func convert(rate:Double) -> Double {
  var total: Double = 0
  // slices the memorized number
  for stringNumber in stringNumbers {
    //Convert string into Int to calculate
    if let number = Double(stringNumber) {
      total = number * rate
    }
  }
  clear()
  return total
}
```

Autodetection du langage

Implementation graphique

L'autodetection est activée via un switch sur l'onglet traduction



Quand celui ci est activé, la langue du pays de départ est changée en fonction du texte entré dans le textField Input.

Model

La fonction detectLanguage() est implémentée dans le model translationService.

Cette fonction prend en paramètre une string : le texte à traduire.

Celle-ci repose sur une requete vers L'API google translate avec les paramètres suivants :

```
guard let urlEncodedText = string.addingPercentEncoding(withAllowedCharacters: .ur
lHostAllowed) else {return}
    let request = "https://translation.googleapis.com/language/translate/v2/detect
?key=(TranslationService.API_KEY)&q=(urlEncodedText)"
```

Le guard statement permet d'éviter une url non valide.

La requete est effectuée grâce à Alamofire et dispatchée sur une background queue afin de ne pas ralentir l'affichage de l'UI.

ce dernier 'escape'/retourne une string représentant le langage reconnu.