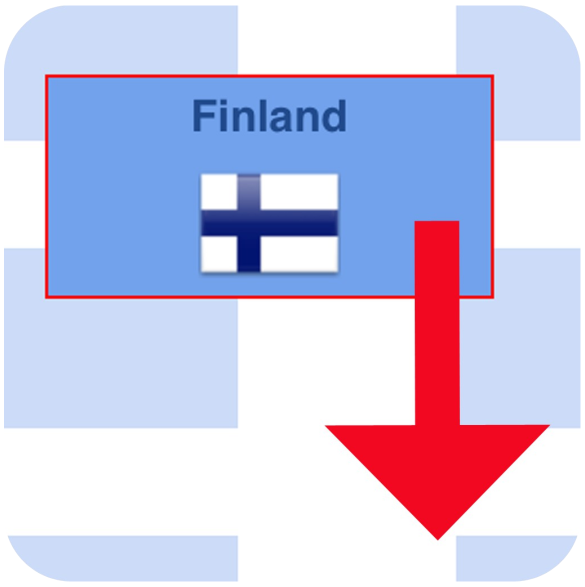


# IOS Drag&Drop Framework

# ArraSolta Manual

Prepared by: Ulrich Vormbrock
Prepared for: IOS developers
Date: 02.11.2015

Apfelstrudel Web
strudel goes creative!

## Table of Contents

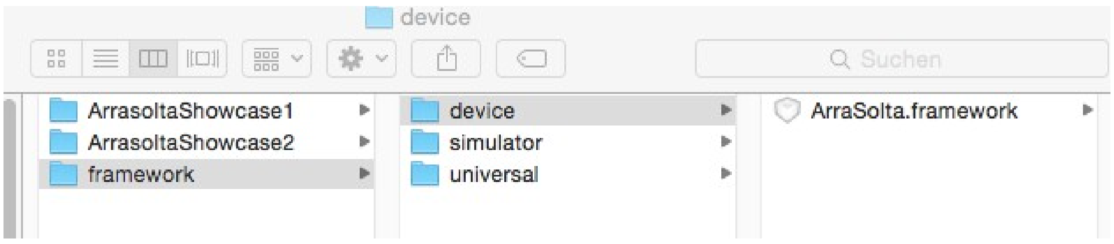
## 1 Prerequisites

**ArraSolta** is written in Objective C, thus it's intended for Xcode projects written in Objective C. In addition, **ArraSolta** is not based on storyboards, but on pure Objective C code. As **ArraSolta** works with auto layout, device rotation and varying screen size are not an issue.

The framework files of **ArraSolta** are delivered in three different versions:

1. Device (***arm64***, ***armv6***, ***armv7s***) – release version for the iPhone and iPad (788 kB),
2. Simulator (***i386***) – for running on simulator only (450 kB),
3. Universal – for running both on simulator, both on physical devices (1.2 MB).

Below please find the directory structure of the framework delivery:



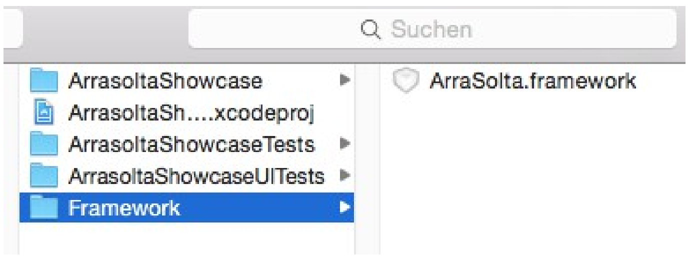
Both Xcode sample projects (***ArraSoltaShowcase1*** and ***ArrasoltaShowcase2*** – delivered together with **ArraSolta**) use the universal version of this framework – thus you can run and test the sample projects both with the simulator, both with a physical device.

**ArraSolta** requires IOS 8.0 or above.
**ArraSolta** is based on UICollectionView with UICollectionViewFlowLayout.

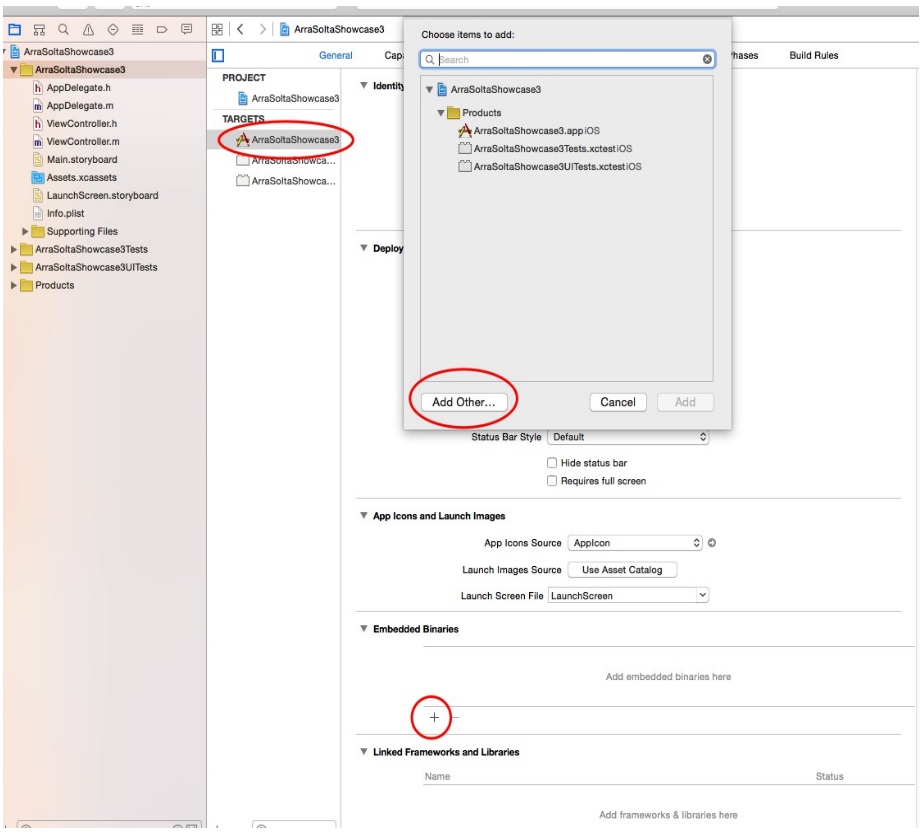
## 2 Integrating ArraSolta into your project

In order to become familiar with **ArraSolta**, we recommend that you run and test at least one of the sample Xcode projects which are delivered with **ArraSolta**. Nevertheless, if you want to integrate ArraSolta into your existing Xcode project, you should read the following explanations.

First make sure that you integrate this framework correctly into your Xcode project. We recommend that you first create a subfolder (using the Finder) which can be named ***Framework***. Copy one of the three above-mentioned framework files into your new folder. As already mentioned, use the universal version at the beginning – you can change it later when you plan to perform a final upload to iTunes Connect.

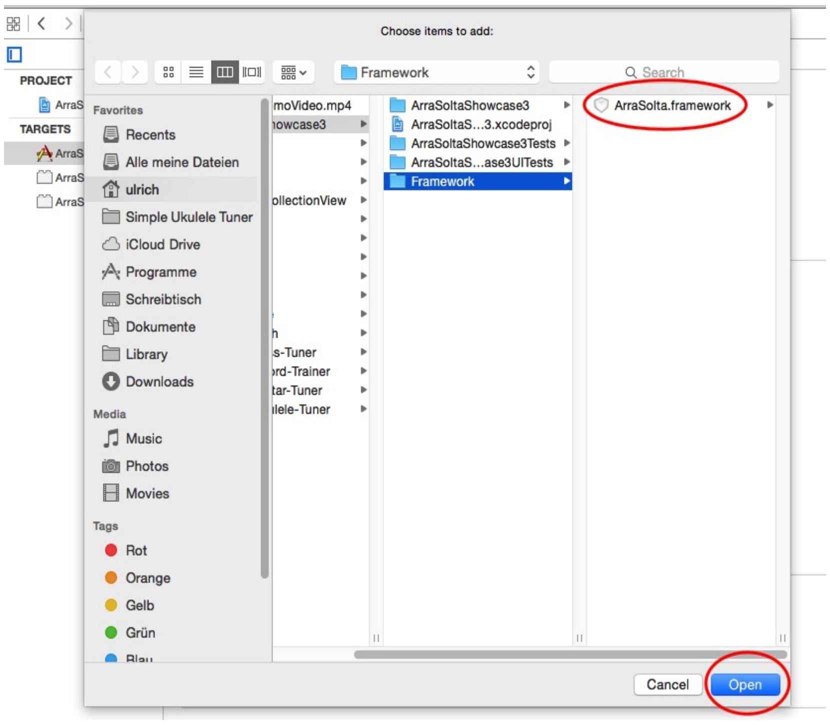


As next step, you need to inform Xcode about the new framework. Just go to ***Targets*** → ***Embedded Binaries*** → ***Add Other:***

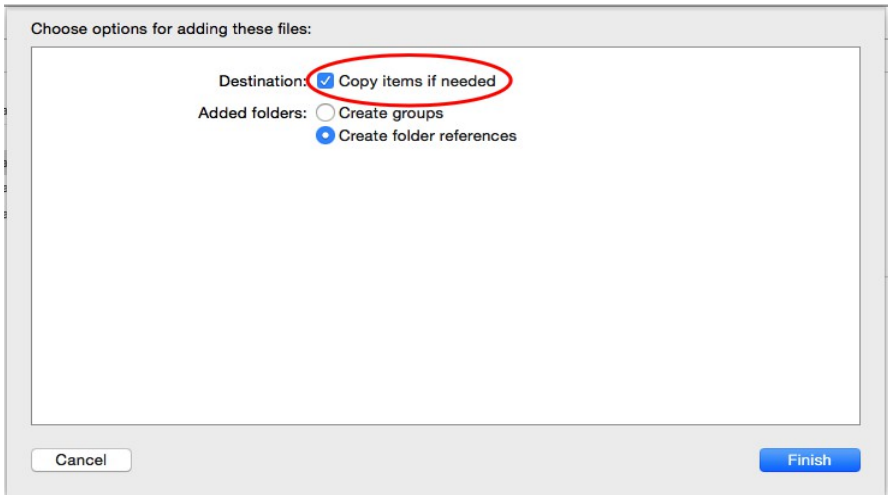

Choose items to add:
ArraSoltaShowcase3
Products
ArraSoltaShowcase3.app iOS
ArraSoltaShowcase3Tests.xctest iOS
ArraSoltaShowcase3UITests.xctest iOS
Add Other...
Cancel
Add
Embedded Binaries
Add embedded binaries here
Linked Frameworks and Libraries
Add frameworks & libraries here

It's important that you choose ***Embedded Binaries*** and NOT ***Linked Framework and Libraries***.

Now choose the ***ArraSolta.framework*** file from your subfolder (you created):



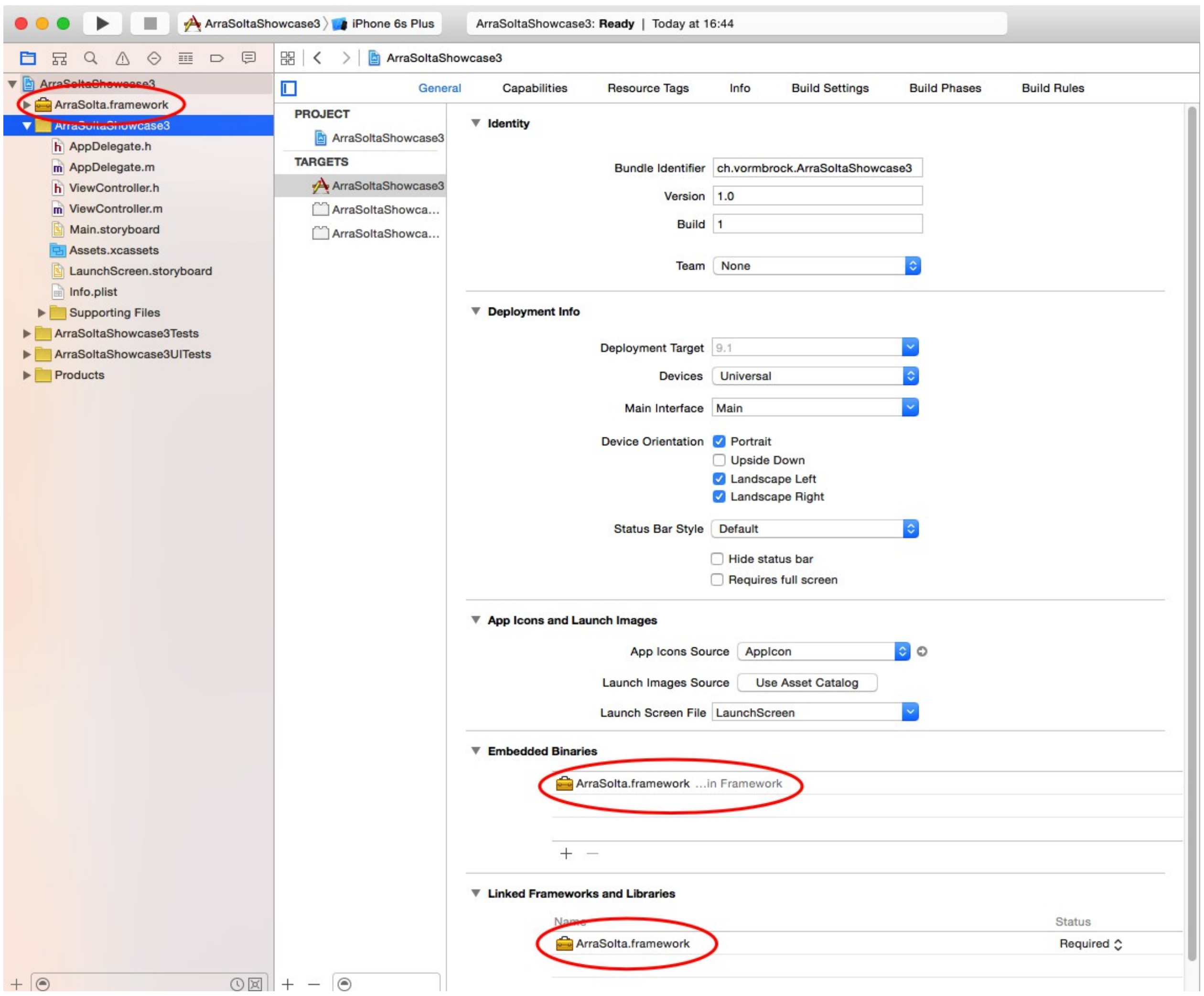
Please watch out that under ***Destination*** the Option ***Copy items if needed*** is enabled:


Choose options for adding these files:

Destination: [✓ Copy items if needed]

Added folders: Create groups [✓ Create folder references]

If you have successfully integrated **ArraSolta** into your project, you should see a screen like this:



Identity

Bundle Identifier: ch.vormbrock.ArraSoltaShowcase3

Version: 1.0

Build: 1

Team: None

Deployment Info

Deployment Target: 9.1

Devices: Universal

Main Interface: Main

Device Orientation: [✓] Portrait, [ ] Upside Down, [✓] Landscape Left, [✓] Landscape Right

Status Bar Style: Default

[ ] Hide status bar

[ ] Requires full screen

App Icons and Launch Images

App Icons Source: AppIcon

Launch Images Source: Use Asset Catalog

Launch Screen File: LaunchScreen

Embedded Binaries

ArraSolta.framework …in Framework

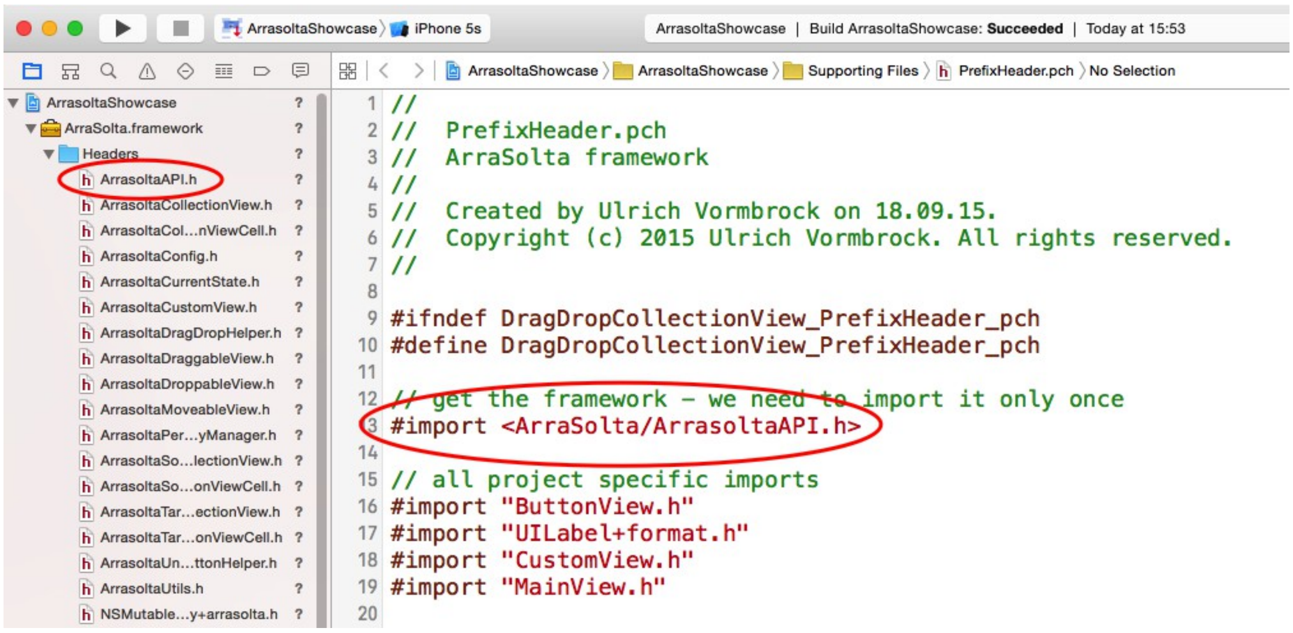
Linked Frameworks and Libraries

| Name | Status |
|---|---|
| ArraSolta.framework | Required |

Now the integration part is done – now you need to perform an import of the API.

## 3 Organizing Imports – ArrasoltaAPI.h

This framework contains more than 20 classes – but you only need to import one master header file named ***ArrasoltaAPI.h*** - which in return contains further imports:


ArrasoltaShowcase 〉 iPhone 5s
ArrasoltaShowcase | Build ArrasoltaShowcase: Succeeded | Today at 15:53
ArrasoltaShowcase 〉 ArrasoltaShowcase 〉 Supporting Files 〉 PrefixHeader.pch 〉 No Selection
ArrasoltaShowcase
ArraSolta.framework
Headers
ArrasoltaAPI.h
ArrasoltaCollectionView.h
ArrasoltaCol...nViewCell.h
ArrasoltaConfig.h
ArrasoltaCurrentState.h
ArrasoltaCustomView.h
ArrasoltaDragDropHelper.h
ArrasoltaDraggableView.h
ArrasoltaDroppableView.h
ArrasoltaMoveableView.h
ArrasoltaPer...yManager.h
ArrasoltaSo...lectionView.h
ArrasoltaSo...onViewCell.h
ArrasoltaTar...ectionView.h
ArrasoltaTar...onViewCell.h
ArrasoltaUn...ttonHelper.h
ArrasoltaUtils.h
NSMutable...y+arrasolta.h

```
//
//  PrefixHeader.pch
//  ArraSolta framework
//
//  Created by Ulrich Vormbrock on 18.09.15.
//  Copyright (c) 2015 Ulrich Vormbrock. All rights reserved.
//

#ifndef DragDropCollectionView_PrefixHeader_pch
#define DragDropCollectionView_PrefixHeader_pch

// get the framework - we need to import it only once
#import <ArraSolta/ArrasoltaAPI.h>

// all project specific imports
#import "ButtonView.h"
#import "UILabel+format.h"
#import "CustomView.h"
#import "MainView.h"

```

Put the following line of code in all files which deal with the ArraSolta framework, as in the View Controller, in the UIView containing the drag&drop functionalities and into your custom UIView which should be draggable and droppable:

**#import <ArraSolta/ArrasoltaAPI.h>**

In order to avoid naming conflicts, this framework uses the ***Arrasolta*** prefix in all its classes and also in the reuse identifiers of the collection view cells!

It's recommended to **build** your project in order to make sure that ArraSolta is correctly integrated!

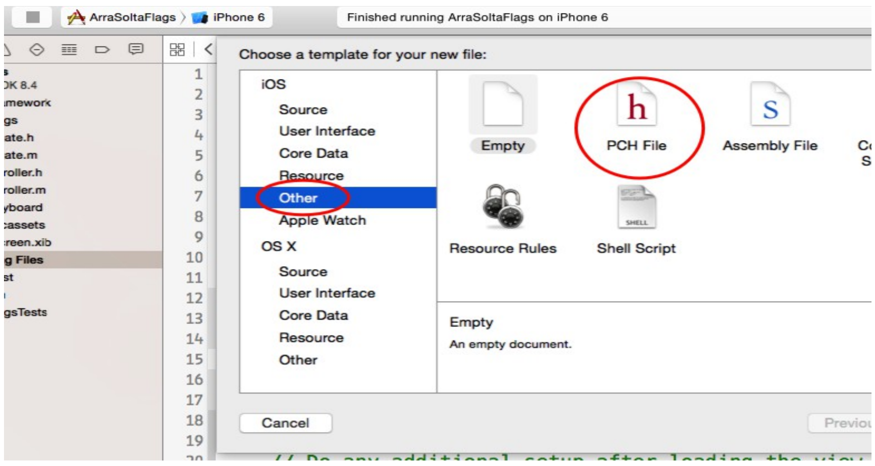
**Hint:**

Instead of importing **<ArraSolta/ArrasoltaAPI.h>** into multiple files, you can also generate a central  PCH file and import it once there.

### 3.1 Generating a central PCH file

Although it's not subject of this manual, we like to show you how to generate a PCH file. It will facilitate further implementation steps, as you'll see later.

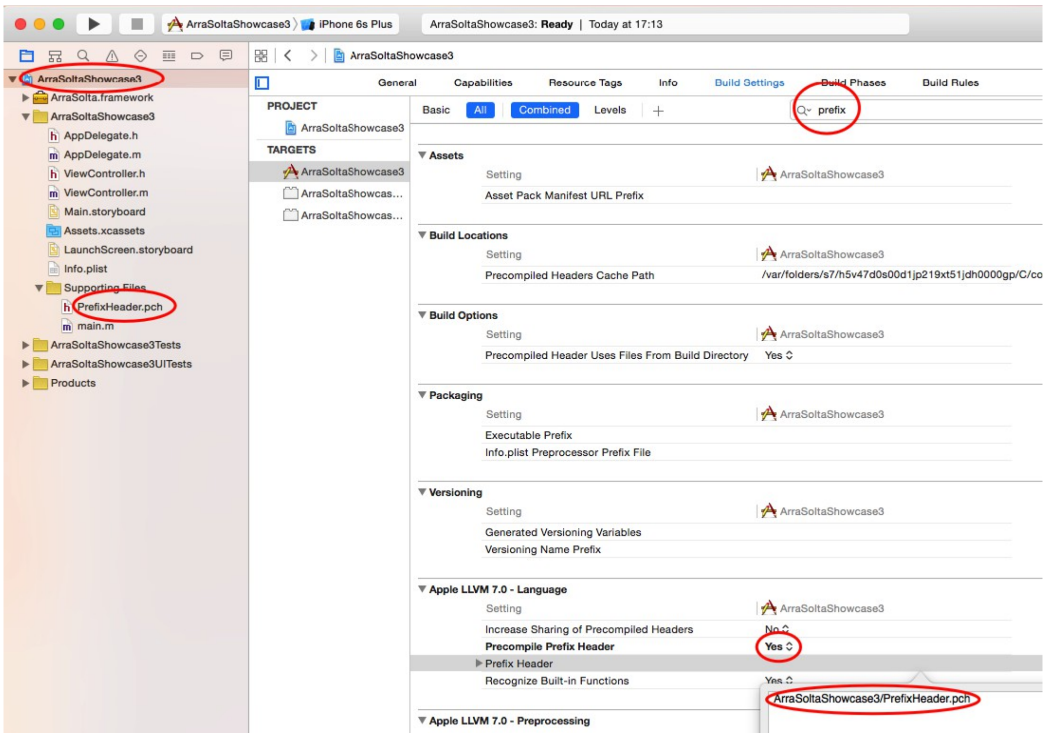
Just right-click at one project file (best practice: right-click at the folder „Supporting Files“ if you want to have the PCH file there) and choose „Other“ and „PCH File“. By default, the PCH file will be named „PrefixHeader“. You can also choose another name for it – maybe in the case that you want to work with multiple PCH files.



Unfortunately, Xcode doesn't add this file to the build settings automatically – you need to do it manually.

Just go to ***Build Settings*** and make a pre-search with the term ***prefix*** in order to find the option ***prefix header*** more easily. Now it's crucial that you change ***Precompile Prefix Header*** from *NO* to ***YES***! Then enter the path and the name of the PCH file below into ***Prefix Header***. That's the most error-prone part: make sure that you enter the project name (here: ) as well as the name and the extension of the PCH file, separated by a slash – thus something like this:

**ArraSoltaShowcase3/PrefixHeader.pch**



Assets
Setting | ArraSoltaShowcase3
Asset Pack Manifest URL Prefix

Build Locations
Setting | ArraSoltaShowcase3
Precompiled Headers Cache Path | /var/folders/s7/h5v47d0s00d1jp219xt51jdh0000gp/C/co

Build Options
Setting | ArraSoltaShowcase3
Precompiled Header Uses Files From Build Directory | Yes

Packaging
Setting | ArraSoltaShowcase3
Executable Prefix
Info.plist Preprocessor Prefix File

Versioning
Setting | ArraSoltaShowcase3
Generated Versioning Variables
Versioning Name Prefix

Apple LLVM 7.0 - Language
Setting | ArraSoltaShowcase3
Increase Sharing of Precompiled Headers | No
**Precompile Prefix Header** | **Yes**
Prefix Header | ArraSoltaShowcase3/PrefixHeader.pch
Recognize Built-in Functions | Yes

Apple LLVM 7.0 - Preprocessing

Make sure that your PCH file is correctly implemented - transfer your **ArraSolta** imports

**#import <ArraSolta/ArrasoltaAPI.h>**

from all files (Vie Controller and UIViews) to your PCH file and make a build. You should get no compile error!

## 4 Using ArraSolta Singletons Macros

This framework provides three different singletons which are open to the user. These are

1. ***ArrasoltaConfig*** (all configurations concerning layout and drag&drop behaviour),
2. ***ArrasoltaUndoButtonHelper*** (binding the undo/redo button to the framework).

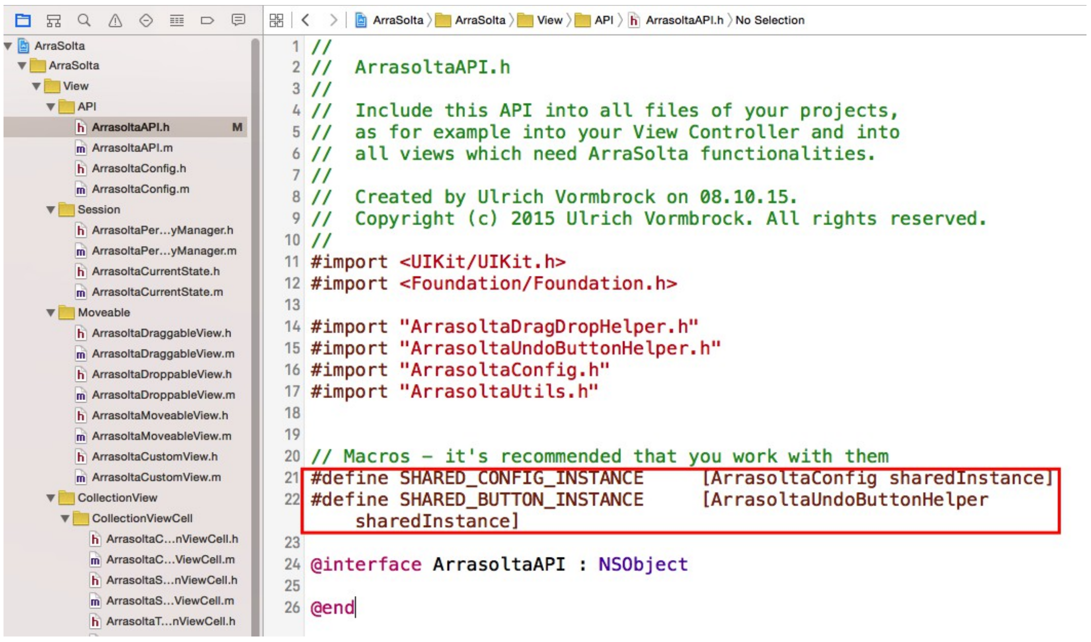
The function of these singletons will be explained more in detail later!

**Please remember:**

| |
|---|
| A singleton is a design pattern which allows only one instance for a given class and which provides a global access point to such instance. Singletons are an easy and powerful way to share data between different parts of a project, without the need to pass objects manually from one class to another. |

As an IOS app only needs to satisfy the requirements of one user, all configurations can be passed to and from a singleton instance, as for example the line-spacing of a collection view or the width-height ratio of a draggable item (custom UIView).

The macros (which are open to the user) are already defined in ***ArrasoltaAPI.h*** **–** thus you won't need to define them in your files:



```
//
//  ArrasoltaAPI.h
//
//  Include this API into all files of your projects,
//  as for example into your View Controller and into
//  all views which need ArraSolta functionalities.
//
//  Created by Ulrich Vormbrock on 08.10.15.
//  Copyright (c) 2015 Ulrich Vormbrock. All rights reserved.
//
#import <UIKit/UIKit.h>
#import <Foundation/Foundation.h>

#import "ArrasoltaDragDropHelper.h"
#import "ArrasoltaUndoButtonHelper.h"
#import "ArrasoltaConfig.h"
#import "ArrasoltaUtils.h"


// Macros - it's recommended that you work with them
#define SHARED_CONFIG_INSTANCE      [ArrasoltaConfig sharedInstance]
#define SHARED_BUTTON_INSTANCE      [ArrasoltaUndoButtonHelper
    sharedInstance]

@interface ArrasoltaAPI : NSObject

@end
```

We recommend to use these macros (instead of redefining them) – see example:

```
[SHARED_CONFIG_INSTANCE setMinInteritemSpacing:10];
[SHARED_CONFIG_INSTANCE setMinLineSpacing:6];

[SHARED_CONFIG_INSTANCE setBackgroundColorSourceView:[UIColor colorWithRed:0.89
    green:0.92 blue:0.98 alpha:1.0]];
[SHARED_CONFIG_INSTANCE setBackgroundColorTargetView:[UIColor colorWithRed:0.89
    green:0.92 blue:0.98 alpha:1.0]];
```

# 5 Programming with ArraSolta

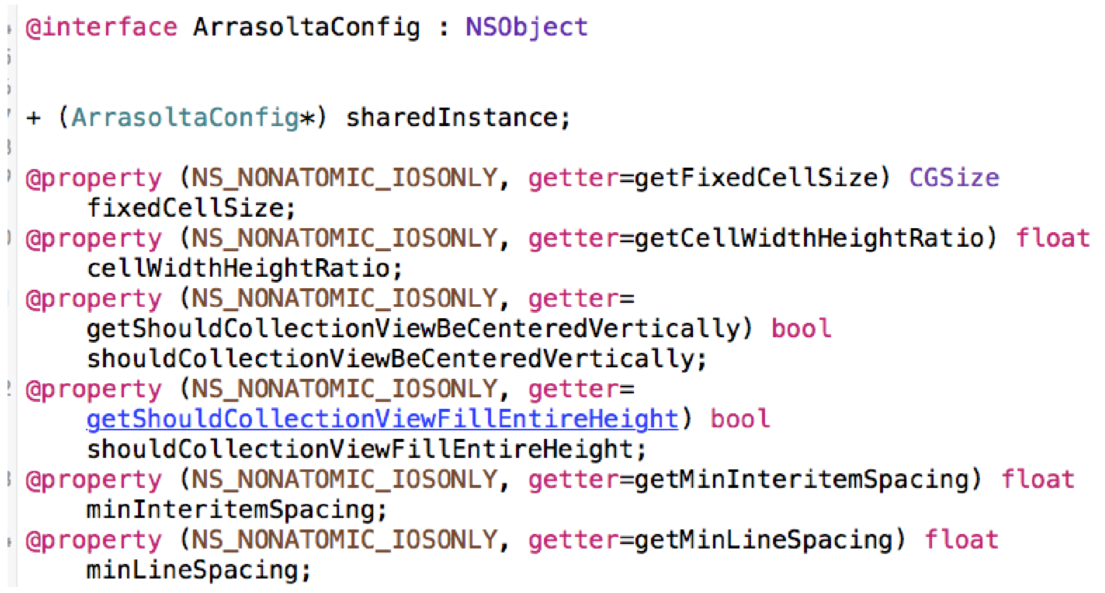
## 5.1 View Controller

The View Controller is the entry-point for the **ArraSolta** framework, as you perform all required configurations concerning

1. the collection views (one for the source and another one for the target items),
2. the draggable views (customizable UIViews which may contain further subviews) and
3. the data source of the source collection view (mutable dictionary with all custom draggable views – properties must be set before adding those views to the dictionary!).

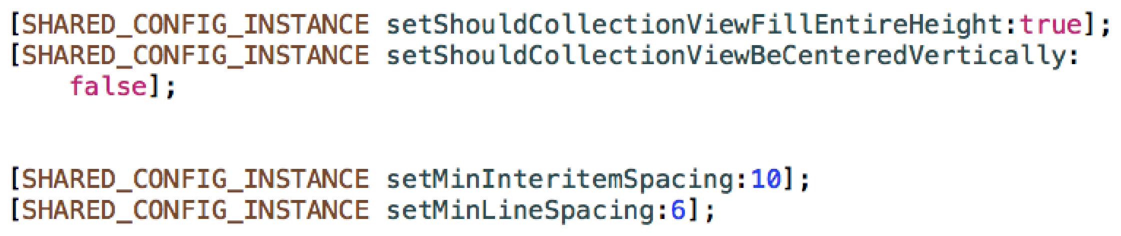
As precondition, your custom draggable UIView class must be created before! In the View Controller, you need to generate instances of your custom view, setting the properties and then adding all these instances to your Mutable Dictionary (which serves as data source).

When you look at the ***ArrasoltaConfig*** interface, you'll discover **about 25 properties** which can be set in the View Controller, as for example ***minInterimSpacing*** in conjunction with Collection Views – they are explained more in detail in ***7.1 Arrasolta Config***:



```
@interface ArrasoltaConfig : NSObject


+ (ArrasoltaConfig*) sharedInstance;

@property (NS_NONATOMIC_IOSONLY, getter=getFixedCellSize) CGSize
    fixedCellSize;
@property (NS_NONATOMIC_IOSONLY, getter=getCellWidthHeightRatio) float
    cellWidthHeightRatio;
@property (NS_NONATOMIC_IOSONLY, getter=
    getShouldCollectionViewBeCenteredVertically) bool
    shouldCollectionViewBeCenteredVertically;
@property (NS_NONATOMIC_IOSONLY, getter=
    getShouldCollectionViewFillEntireHeight) bool
    shouldCollectionViewFillEntireHeight;
@property (NS_NONATOMIC_IOSONLY, getter=getMinInteritemSpacing) float
    minInteritemSpacing;
@property (NS_NONATOMIC_IOSONLY, getter=getMinLineSpacing) float
    minLineSpacing;
```

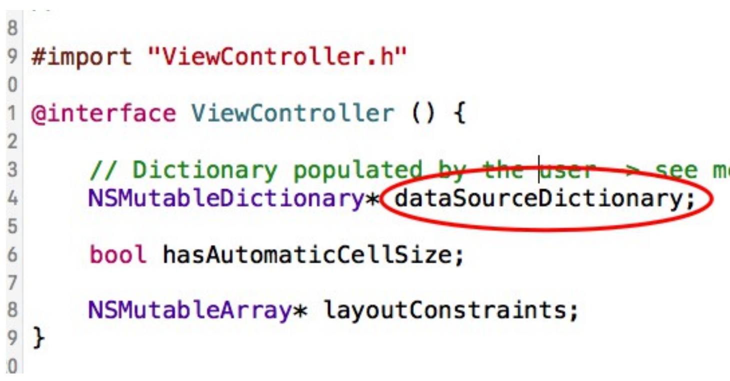
Below please find a possible implementation in the **View Controller**:



```
[SHARED_CONFIG_INSTANCE setShouldCollectionViewFillEntireHeight:true];
[SHARED_CONFIG_INSTANCE setShouldCollectionViewBeCenteredVertically:
    false];


[SHARED_CONFIG_INSTANCE setMinInteritemSpacing:10];
[SHARED_CONFIG_INSTANCE setMinLineSpacing:6];
```

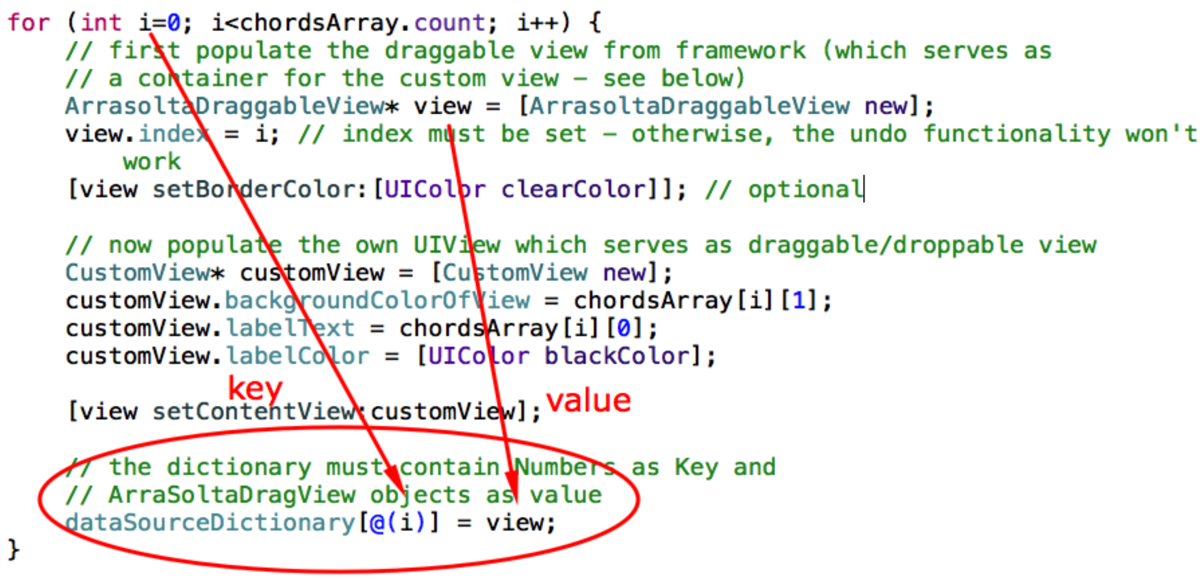
As mentioned before, you need to populate the Source Collection View with your custom (draggable/droppable) elements.

First you need to declare a Mutable Dictionary as private member – for example like this:



```
#import "ViewController.h"

@interface ViewController () {

    // Dictionary populated by the user -> see meth
    NSMutableDictionary* dataSourceDictionary;

    bool hasAutomaticCellSize;

    NSMutableArray* layoutConstraints;
}

```

After instantiating the Mutable Dictionary (please don't forget!), you should first wrap **all instances of your custom UIViews** into **ArrasoltaDraggableViews** and then add all these wrapper instances to the newly created Mutable Dictionary – see the following example:



```
for (int i=0; i<chordsArray.count; i++) {
    // first populate the draggable view from framework (which serves as
    // a container for the custom view - see below)
    ArrasoltaDraggableView* view = [ArrasoltaDraggableView new];
    view.index = i; // index must be set - otherwise, the undo functionality won't
        work
    [view setBorderColor:[UIColor clearColor]]; // optional

    // now populate the own UIView which serves as draggable/droppable view
    CustomView* customView = [CustomView new];
    customView.backgroundColorOfView = chordsArray[i][1];
    customView.labelText = chordsArray[i][0];
    customView.labelColor = [UIColor blackColor];

    [view setContentView:customView];

    // the dictionary must contain Numbers as Key and
    // ArraSoltaDragView objects as value
    dataSourceDictionary[@(i)] = view;
}
```

**Important:**

The Mutable Dictionary (as data source for the Source Collection View) must contain a **NSNumber** object (starting from 0) as **key** and an instance of **ArrasoltaDraggableView** as **value**. **Warning**: don't add your custom view directly to the Mutable Dictionary! You can learn more about the relation between the framework classes in the chapter ***6 Architecture of the ArraSolta framework***.

***ArrasoltaDraggableView*** is part of the **ArraSolta** framework and serves as container for your custom draggable view. It can contain a border (width and colour) – for populating ***ArrasoltaDraggableView with your own custom view,*** you need to call the following method:

***[view setContentView:customView];***

## 5.2 Main View containing the collection views

You need an UIView which contains the two collection views (source and target) and an optional undo/redo button.

This framework is shipped with two different sample projects – thus you can get a first insight how to program a view with drag and drop functionality.

As you can see in the sample projects, there is a certain amount of source code in the main view, as the following tasks should be done:

1. Build all views (headline, undo/redo button, collection views) and add them to the main view,
2. Fill the collection view's delegate methods with code (***UICollectionViewDataSource*** and ***UICollectionViewDelegate***),
3. Initialize the drag and drop helper class ***ArrasoltaDragDropHelper***.
4. Setup the layout constraints, as we work with auto layout,
5. Handle device rotation with the aid of ***NSNotificationCenter - UIDeviceOrientationDidChangeNotification***.

We expect that you are familiar with auto layout – thus we explain the first two points only. Anyhow, you can see in the sample projects how step 3. and 4. can be done.

In the sample projects, we have put all methods into a single class (MainView), but in order to make your code more readable (“separation of concerns”), you can create a superclass of MainView and put all the code concerning layout constraints in it.

Now we explain step-by-step how to build the collection views and how to handle their delegates.

### 5.2.1 Source Collection View

The source collection view must be populated by a ***NSDictionary*** within the initialisation process ***initWithFrame.*** Remember that the source collection view must be filled***!*** Let's have a look into one sample project:

```
// Prepare source collection view
self.sourceItemsDictionary = [SHARED_CONFIG_INSTANCE getSourceItemsDictionary]
    ;
self.numberOfSourceItems = (int)self.sourceItemsDictionary.count;

self.sourceCollectionView = [[ArrasoltaSourceCollectionView alloc]
    initWithFrame:frame withinView:self];
```

As you cans see, the dictionary (previously defined in the ViewController) is read from the ***ArrasoltaConfig*** singleton – see macro ***SHARED_CONFIG_INSTANCE***. We also need the number of items which simply is the number of dictionary elements. The instantiation of the source collection view is easy: please use the class ***ArrasoltaSourceCollectionView*** and pass the current frame and the embedding view (=self) as arguments.

### 5.2.2 Target Collection View

Contrary to the source collection view, the target collection view doesn't need be populated at the beginning: it will be populated later when the user drags and drops elements into it. Anyhow, if you work with Core Data, you should populate the target collection view with the elements from the previous session. Let's see how to do:

```
    // Prepare target collection view - it's still empty!
    self.targetItemsDictionary = [NSMutableDictionary new];
    self.numberOfTargetItems = [SHARED_CONFIG_INSTANCE getNumberOfTargetItems];

    self.targetCollectionView = [[ArrasoltaTargetCollectionView alloc]
        initWithFrame:frame withinView:self sourceDictionary:self.
        sourceItemsDictionary targetDictionary:self.targetItemsDictionary];
```

Contrary to the source collection view, the target collection view ***ArrasoltaTargetCollectionView*** is instantiated with two further arguments – namely with the source and with the target dictionary.

The number of items comes from ***ArrasoltaConfig:*** if source items are consumable, the framework sets it equal to the number of source items – if source items are not consumable, the value must be set in the View Controller. In the first case, it wouldn't make sense to set the number of target elements different than the number of source elements!

### 5.3.3 UICollectionViewDataSource Protocol

According to Apple's specification, all data source objects must at least implement the **numberOfItemsInSection** and **cellForItemAtIndexPath** methods! Lets start with the first one:

```
#pragma mark <UICollectionViewDataSource>
-(NSInteger)collectionView:(UICollectionView *)collectionView
    numberOfItemsInSection:(NSInteger)section {

    if ([collectionView isKindOfClass:[ArrasoltaSourceCollectionView class
        ]]) {
        return self.numberOfSourceItems;
    } else {
        return self.numberOfTargetItems;
    }
}
```

The implementation of ***numberOfItemsInSection*** is trivial: we first make a class check of ***collectionView*** (***ArrasoltaSourceCollectionView*** and ***ArrasoltaTargetCollectionView***) and return the number of items which in return represent the number of items in the correspondent data source object (***NSDictionary***).

The implementation of ***cellForItemAtIndexPath*** is easy as well:

```
-(ArrasoltaCollectionViewCell *)collectionView:(UICollectionView *)
    collectionView cellForItemAtIndexPath:(NSIndexPath *)indexPath {

    // Important: order of dictionaries must be maintained:
    // 1. source dictionary
    // 2. target dictionary
    return [ArrasoltaUtils getCell:collectionView forIndexPath:indexPath
        cellDictionaries:@[self.sourceItemsDictionary, self.
        targetItemsDictionary]];
}
```

Thanks to our utility helper class ***ArrasoltaUtils***, we only need one line of code inside ***cellForItemAtIndexPath***. Please pay attention to the order of the dictionaries (third argument – array with source and target dictionary)!

Remember that ***cellForItemAtIndexPath*** is responsible for creating, configuring and returning the appropriate cell for the given item (see Apple's specification).

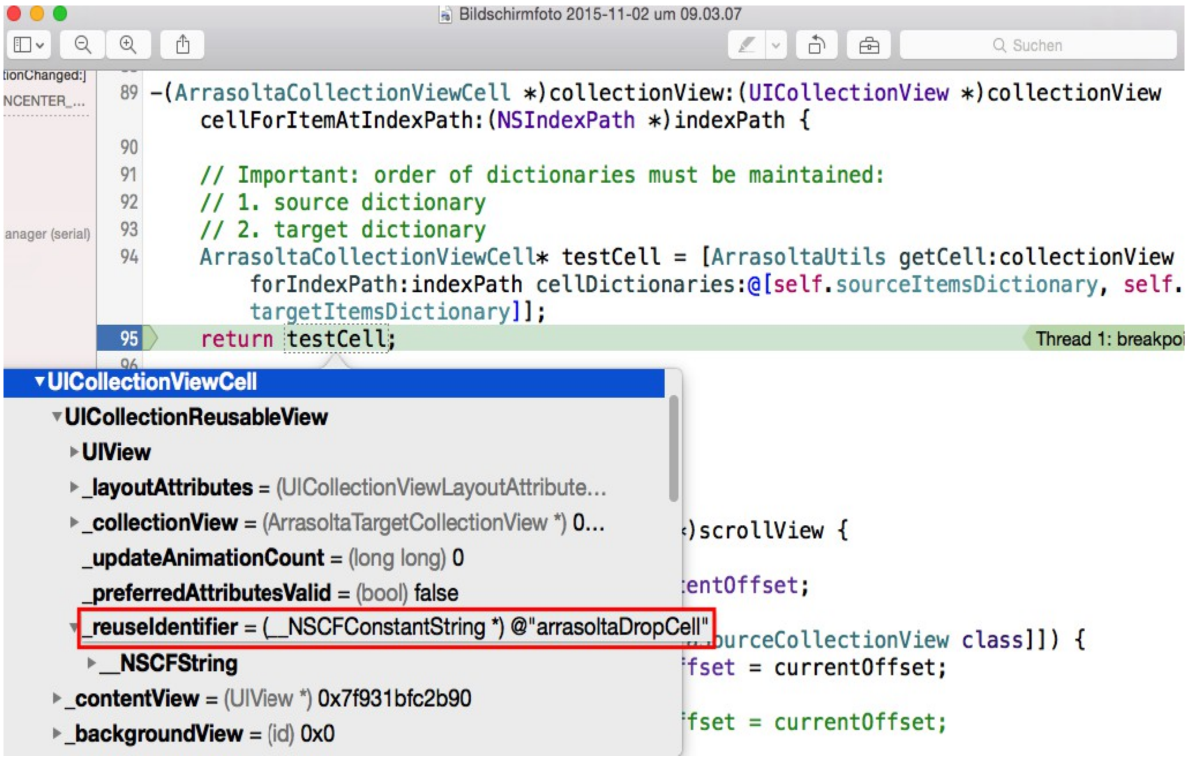
**Attention:**

Please take into account that the returned **cell** object is not of type ***UICollectionViewCell***, but of ***ArrasoltaCollectionViewCell*** (which is part of this framework). If you set UICollectionViewCell instead, you get a compile error!

**Note:**

If you are familiar with collection views, you might wonder why we don't use a **reuse identifier** for the **cells**. Such reuse identifier exist for both collection views, but they are handled internally within the framework. As already mentioned, both reuse identifiers have ***arrasolta*** as prefix – so there is no danger of naming conflicts.

To make sure that the **reuse identifiers** are really set, you can set a breakpoint as shown in the screenshot below and analyse the ***ArrasoltaCollectionViewCell*** object:



```
-(ArrasoltaCollectionViewCell *)collectionView:(UICollectionView *)collectionView
    cellForItemAtIndexPath:(NSIndexPath *)indexPath {

    // Important: order of dictionaries must be maintained:
    // 1. source dictionary
    // 2. target dictionary
    ArrasoltaCollectionViewCell* testCell = [ArrasoltaUtils getCell:collectionView
        forIndexPath:indexPath cellDictionaries:@[self.sourceItemsDictionary, self.
        targetItemsDictionary]];
    return testCell;
```

```
▼UICollectionViewCell
  ▼UICollectionReusableView
    ▶UIView
    ▶_layoutAttributes = (UICollectionViewLayoutAttribute...
    ▶_collectionView = (ArrasoltaTargetCollectionView *) 0...
     _updateAnimationCount = (long long) 0
     _preferredAttributesValid = (bool) false
    ▼_reuseIdentifier = (__NSCFConstantString *) @"arrasoltaDropCell"
      ▶__NSCFString
  ▶_contentView = (UIView *) 0x7f931bfc2b90
  ▶_backgroundView = (id) 0x0
```

```
)scrollView {
entOffset;
urceCollectionView class]]) {
fset = currentOffset;
fset = currentOffset;
```

## 5.3.4 UICollectionViewDelegateFlowLayout Protocol

The ***UICollectionViewDelegateFlowLayout*** protocol provides methods which define the size of items and the spacing between items in the grid. Please note that we work with flow layout, which is a line-based braking layout: as soon as a line is full, a new line is created.

You might wonder why in both Xcode example projects the method ***collectionView:layout:sizeForItemAtIndexPath:*** does not appear! Remember: normally, this method must be implemented, as it describes the ***size*** (height and width) of each cell. But in this case, the framework handles this method internally – thus you won't need to implement it.

The reason why this method is handled internally is simple: we always need to re-calculate the cell sizes as soon as the user

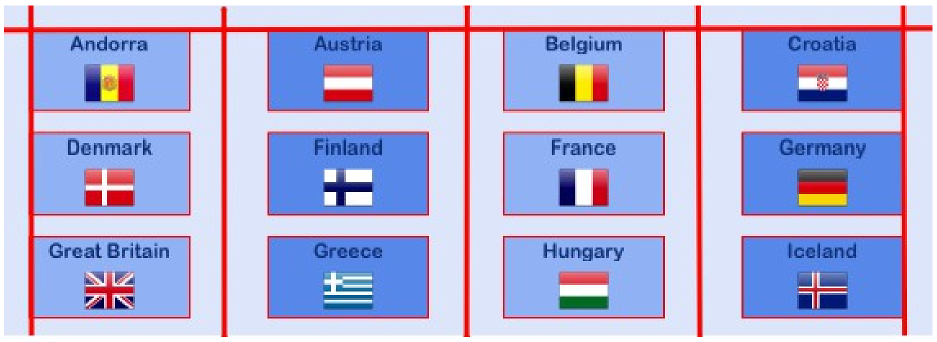
1. performs a device rotation and
2. zooms in and out.

Thus the user is free from the burden of solving resize issues!

Although not implemented in both sample projects,  you can implement the protocol method ***collectionView:layout:insetForSectionAtIndex as well***:

```
#pragma mark <UICollectionViewDelegateFlowLayout>
- (UIEdgeInsets)collectionView:(UICollectionView *)collectionView layout:
    (UICollectionViewLayout*)collectionViewLayout insetForSectionAtIndex:(NSInteger)
    section {

    float paddingTop    = 10.0;
    float paddingLeft   = 10.0;
    float paddingBottom =  0.0;
    float paddingRight  = 10.0;

    return UIEdgeInsetsMake(paddingTop, paddingLeft, paddingBottom, paddingRight);
}
```

As a result, you might get something like this:



As you can see, the cells have a top margin to the header and a left/right spacing between each others. Please note that paddingTop refers to the first row only – there are no top margins between the rows!

**Attention:**

The method ***collectionView:layout:insetForSectionAtIndex*** has no effect if you work with horizontal scrolling! You should set

***[SHARED_CONFIG_INSTANCE setShouldCellOrderBeHorizontal:true];***

in the View Controller if you want to set the insets.

### 5.3.5 Drag Drop Helper

In order to enable drag & drop functionality, you must initialize the ***ArrasoltaDragDropHelper*** class in the init method:

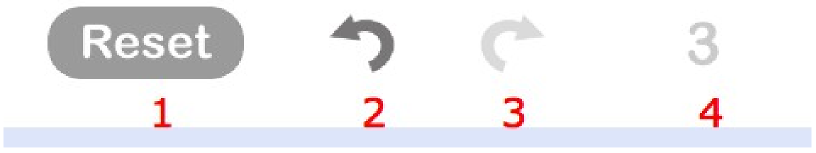
***[[ArrasoltaDragDropHelper sharedInstance] initWithView:self collectionViews:
@[self.sourceCollectionView, self.targetCollectionView] cellDictionaries:
@[self.sourceItemsDictionary, self.targetItemsDictionary]];***

If you forget to initialize it, the collection views will be filled but the items won't be draggable!

## 5.4 Undo / Redo / Reset Button Bar

The undo button is optional – thus you can decide if you want to implement it or not. In both sample projects, you will see how the undo (and redo) button can be integrated into your app.

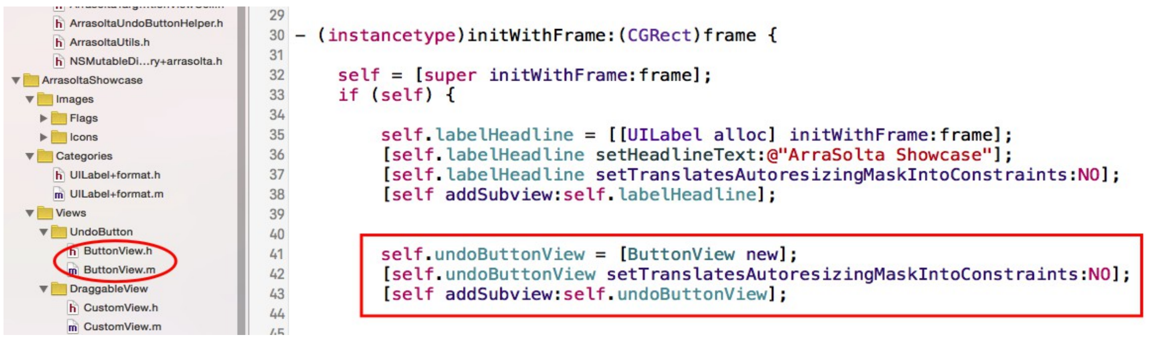
Let's have a look at the components of the button ensemble:



1. **Reset Button**: clears target collection view (and brings all items back to the source collection view),
2. **Undo Button**: reverts all previous actions (such as dropping, deleting, inserting, etc.)
3. **Redo Button**: reverts previous undo action,
4. **Counter**: displays the number of actions from the beginning (touching the undo button, the counter will decrement until zero).

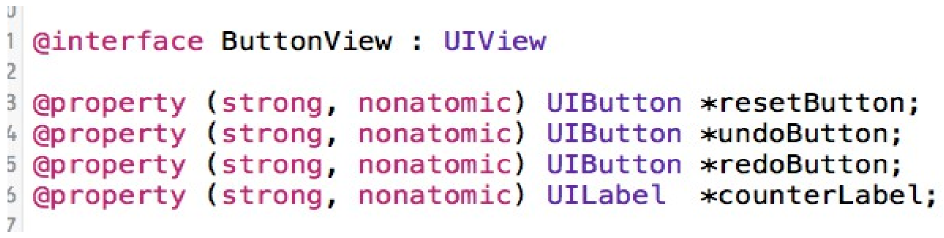
It's up to you to decide which components should be displayed in your app – normally, you won't want to implement the counter – but such counter can be useful for debugging purposes.

We recommend that for implementing such button bar you create a separate UIView – as shown in both sample projects:



```
- (instancetype)initWithFrame:(CGRect)frame {

    self = [super initWithFrame:frame];
    if (self) {

        self.labelHeadline = [[UILabel alloc] initWithFrame:frame];
        [self.labelHeadline setHeadlineText:@"ArraSolta Showcase"];
        [self.labelHeadline setTranslatesAutoresizingMaskIntoConstraints:NO];
        [self addSubview:self.labelHeadline];

        self.undoButtonView = [ButtonView new];
        [self.undoButtonView setTranslatesAutoresizingMaskIntoConstraints:NO];
        [self addSubview:self.undoButtonView];
```

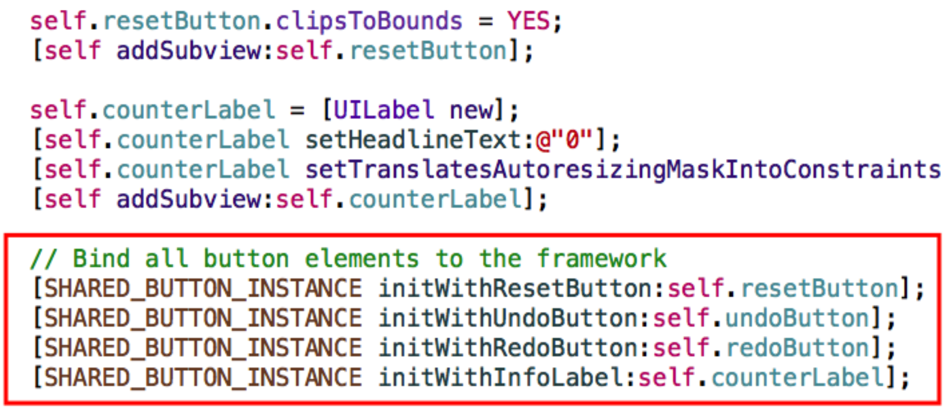
In the public UIView interface, you should declare your button bar components as follows:



```
@interface ButtonView : UIView

@property (strong, nonatomic) UIButton *resetButton;
@property (strong, nonatomic) UIButton *undoButton;
@property (strong, nonatomic) UIButton *redoButton;
@property (strong, nonatomic) UILabel  *counterLabel;

```

You can design your button bar components as you want and even use your own button icons.

The crucial part is the binding of your components to the framework – please don't forget to write the marked lines of code in your UIView class:



```
self.resetButton.clipsToBounds = YES;
[self addSubview:self.resetButton];

self.counterLabel = [UILabel new];
[self.counterLabel setHeadlineText:@"0"];
[self.counterLabel setTranslatesAutoresizingMaskIntoConstraints:NO];
[self addSubview:self.counterLabel];

// Bind all button elements to the framework
[SHARED_BUTTON_INSTANCE initWithResetButton:self.resetButton];
[SHARED_BUTTON_INSTANCE initWithUndoButton:self.undoButton];
[SHARED_BUTTON_INSTANCE initWithRedoButton:self.redoButton];
[SHARED_BUTTON_INSTANCE initWithInfoLabel:self.counterLabel];
```

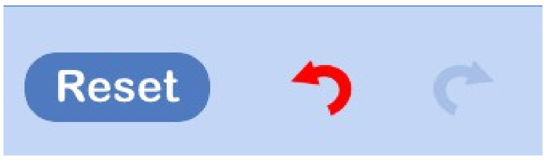
Please note that you won't need to define the macro ***SHARED_BUTTON_INSTANCE*** – as already mentioned in ***4 Using ArraSolta Singletons Macros***, this macro is defined in ***ArrsoltaAPI.h***.

**Hint:**

If you want to use the button icons from the sample projects but with different colours, you can tint them as shown in the following lines of code:

*UIImage* undoBtnImage = [[UIImage imageNamed:@"undo.png"] imageWithRenderingMode:**UIImageRenderingModeAlwaysTemplate**];*
*self.undoButton.**tintColor** = [UIColor **redColor**];*

You will get a red-tinted button icon for the undo button:

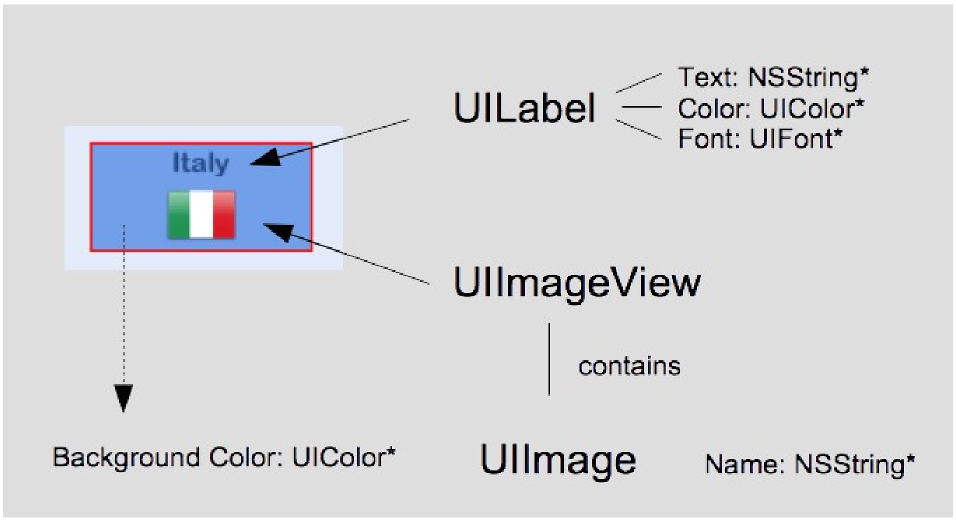


After binding the above-mentioned components to the framework, the functionalities are handled internally – thus you won't need to write a single code concerning tracking of user interactions. The framework automatically makes snapshots of the source and target collection view after each interaction (dropping, inserting, deleting, etc.).

## 5.5 Custom draggable / droppable View

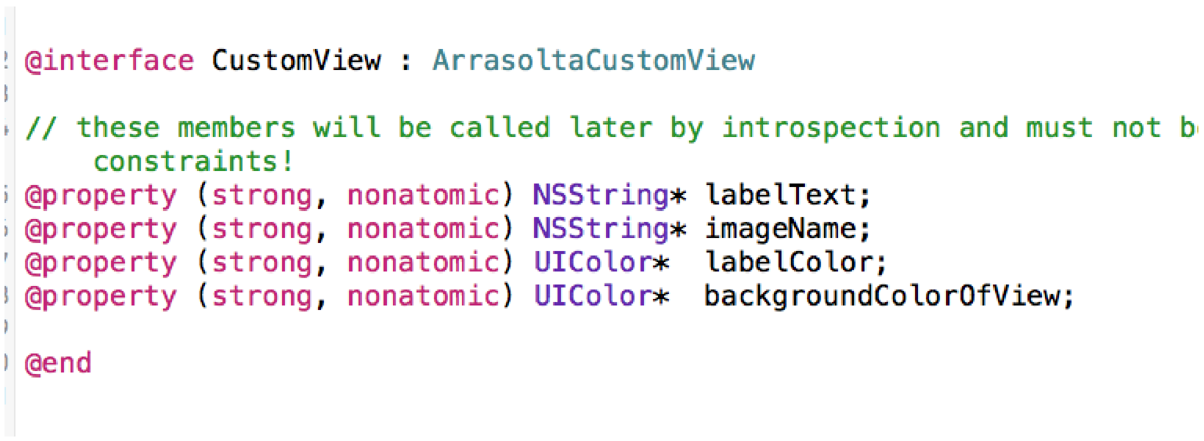
As already mentioned, this framework allows to define own views which can be dragged and dropped between two collection views. Thus you can do much more than only moving images or labels.

A custom draggable view may even contain further subviews. Lets study the draggable view of the first sample project you get with this product:



The composition of this view is simple, as it contains only an UILabel and an UIImageView. Their position is defined by Visual Format Language. We won't explain auto layout issues in detail, as this is not subject of this manual.

Lets have a look at the public interface of such  custom draggable view:



```
@interface CustomView : ArrasoltaCustomView

// these members will be called later by introspection and must not b
    constraints!
@property (strong, nonatomic) NSString* labelText;
@property (strong, nonatomic) NSString* imageName;
@property (strong, nonatomic) UIColor*  labelColor;
@property (strong, nonatomic) UIColor*  backgroundColorOfView;

@end
```

First of all, your custom draggable/droppable view should extend ***ArrasoltaCustomView***.

Probably, you expected that the view components themselves are declared in the public interface, something like this:

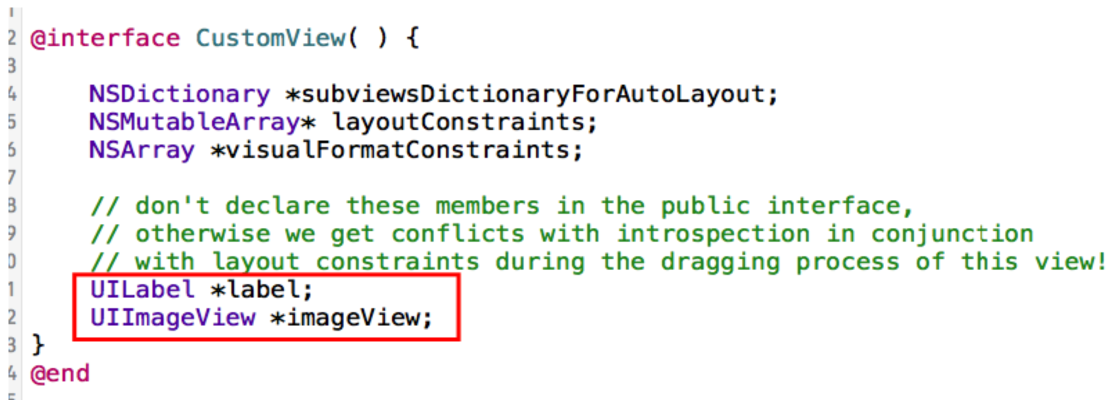
**@property (strong, nonatomic) UIImageView* myImageView;  // forbidden!!!**

**Don't do that!**

The reasons are the following:

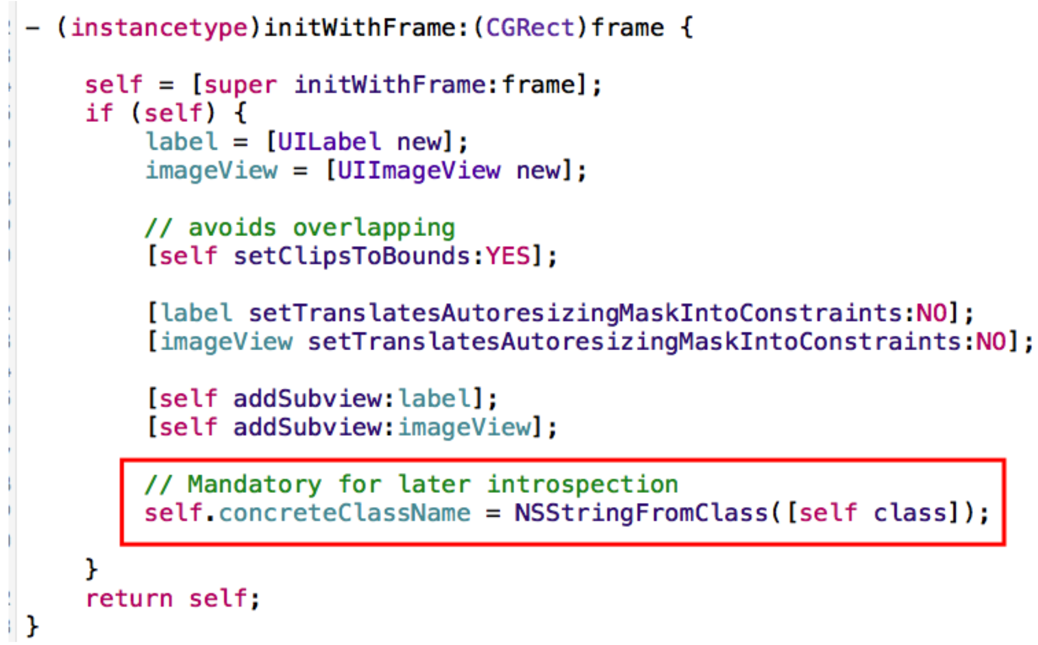
1. The components (as for example UIImageView) are bound to layout constraints (at least if you work with auto layout),
2. As soon as you drop an item into the target collection view, a copy of your custom view is made by introspection,
3. Performing introspection, all layout constraints are corrupted – thus you'll get an exception at run time.

Declare all subview components of your custom view in your private interface and populate them later with the aid of your public setters.



```
@interface CustomView( ) {

    NSDictionary *subviewsDictionaryForAutoLayout;
    NSMutableArray* layoutConstraints;
    NSArray *visualFormatConstraints;

    // don't declare these members in the public interface,
    // otherwise we get conflicts with introspection in conjunction
    // with layout constraints during the dragging process of this view!
    UILabel *label;
    UIImageView *imageView;
}
@end
```

Besides – as prerequisite for introspection, you should set the public member ***concreteClassName*** in the init method of your custom  view - ***concreteClassName*** is a member of ***ArrasoltaCustomView*** which you always should extend:



```
- (instancetype)initWithFrame:(CGRect)frame {

    self = [super initWithFrame:frame];
    if (self) {
        label = [UILabel new];
        imageView = [UIImageView new];

        // avoids overlapping
        [self setClipsToBounds:YES];

        [label setTranslatesAutoresizingMaskIntoConstraints:NO];
        [imageView setTranslatesAutoresizingMaskIntoConstraints:NO];

        [self addSubview:label];
        [self addSubview:imageView];

        // Mandatory for later introspection
        self.concreteClassName = NSStringFromClass([self class]);

    }
    return self;
}
```

If you omit this line of code, the dropped cells will only appear as a dark grey box, missing all components you defined in your custom draggable view.

### 5.4.1 Creating Subview Components

As already mentioned, the subview components (UILabel, UIImageView, etc.) come as private members. You should instantiate and add them (to your custom view) in your init method.

Besides, the layout constraints of your subcomponents must be set!

**Attention:**

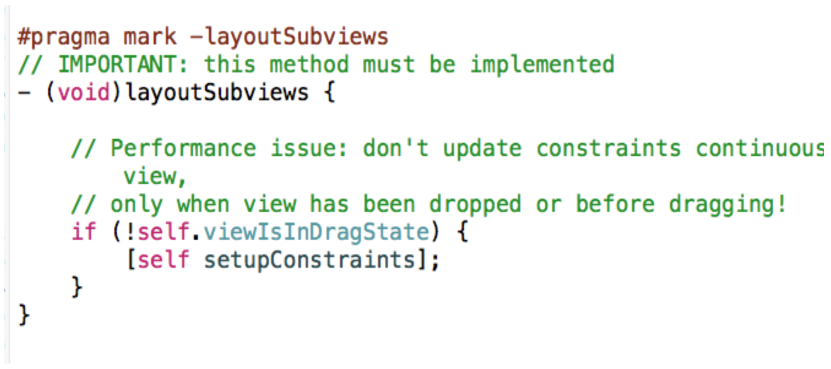
Please take into account that the layout of your components changes

1. during dragging,
2. during zooming and
3. after device rotation.

Let's have a look at what **Apple** writes about layout changes:

> ***layoutSubviews:***
>
> Implement this method if you need more precise control over the layout of your subviews than either the constraint or autoresizing behaviours provide.

We need to implement the ***layoutSubviews*** method:

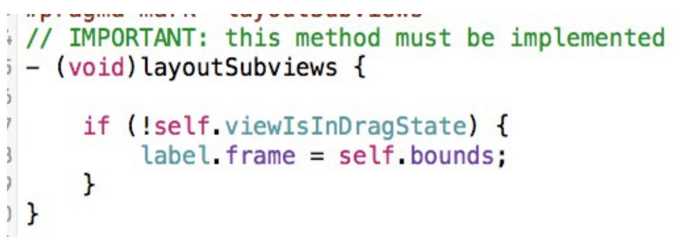


```
#pragma mark -layoutSubviews
// IMPORTANT: this method must be implemented
- (void)layoutSubviews {

    // Performance issue: don't update constraints continuous
        view,
    // only when view has been dropped or before dragging!
    if (!self.viewIsInDragState) {
        [self setupConstraints];
    }
}
```

Don't omit ***layoutSubviews*** – if you do so (and call ***setupConstraints*** directly in your init method), you'll get empty cells!

Please take into account that ***layoutSubviews*** is executed continuously while dragging your custom view. For the sake of performance, the framework (class ***ArrasoltaCustomView*** which you extend) offers the attribute ***viewIsInDragState*** which is set to true as soon as a custom view is moving.

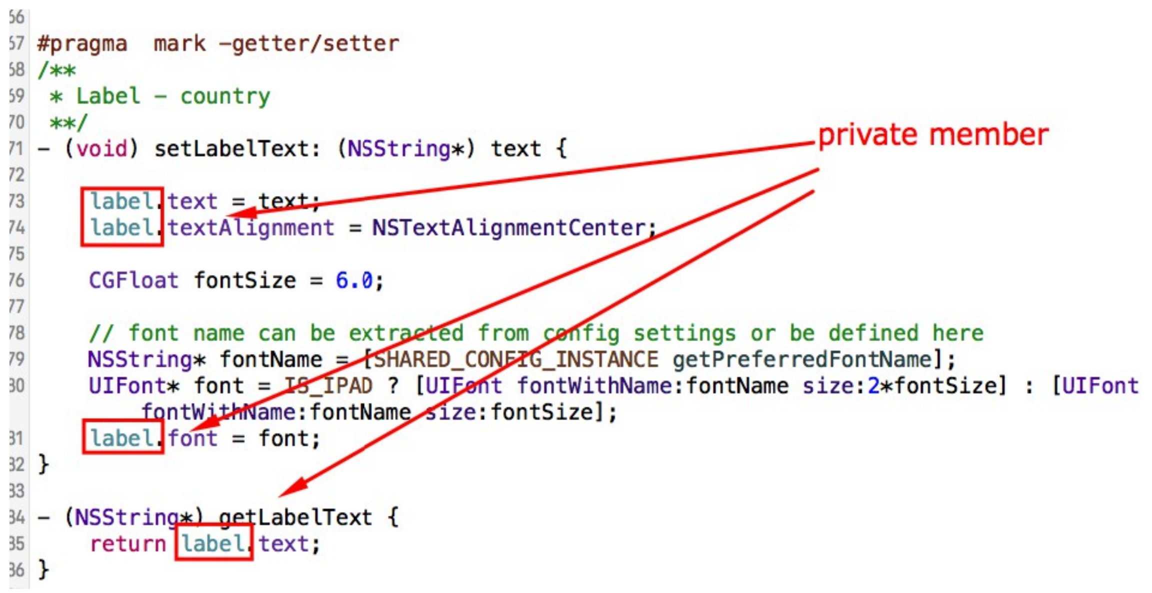
If you don't work with auto layout (see second Xcode sample project), you must update the ***frame*** of your subview components rather than the layout constraints:



```
// IMPORTANT: this method must be implemented
- (void)layoutSubviews {

    if (!self.viewIsInDragState) {
        label.frame = self.bounds;
    }
}
```

### 5.4.2 Writing getters and setters for your subview components

In order to populate your subview components (for example an UILabel with formatted text or an UIImageView with a png file), you should **generate getters and setters** for your public (not private!) members:

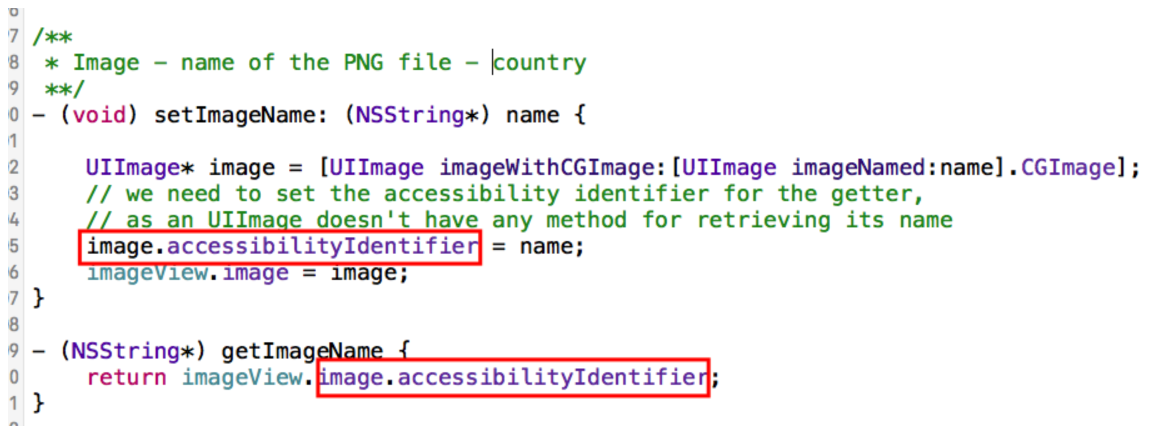


```
#pragma  mark -getter/setter
/**
 * Label - country
 **/
- (void) setLabelText: (NSString*) text {

    label.text = text;
    label.textAlignment = NSTextAlignmentCenter;

    CGFloat fontSize = 6.0;

    // font name can be extracted from config settings or be defined here
    NSString* fontName = [SHARED_CONFIG_INSTANCE getPreferredFontName];
    UIFont* font = IS_IPAD ? [UIFont fontWithName:fontName size:2*fontSize] : [UIFont
        fontWithName:fontName size:fontSize];
    label.font = font;
}

- (NSString*) getLabelText {
    return label.text;
}
```

Just remember:

1. ***labelText*** is a public member and used for later introspection:
   @property (strong, nonatomic) NSString* labelText;
2. ***label*** is an instance of UILabel and a private member of your custom view.
   You populate label indirectly within the setter of the public labelText.

When you populate your image view, watch out for the image name which in the correspondent getter method can't be retrieved – UIImage has no method which returns the image name. Instead you should set the **accessibility identifier** of the image with your image name:



```
/**
 * Image - name of the PNG file - country
 **/
- (void) setImageName: (NSString*) name {

    UIImage* image = [UIImage imageWithCGImage:[UIImage imageNamed:name].CGImage];
    // we need to set the accessibility identifier for the getter,
    // as an UIImage doesn't have any method for retrieving its name
    image.accessibilityIdentifier = name;
    imageView.image = image;
}

- (NSString*) getImageName {
    return imageView.image.accessibilityIdentifier;
}
```

When you have programmed your custom draggable view, don't forget to import it into your **ViewController** (or better in your centralized PCH file).

### 5.4.2 Populating subview components with data (in the View Controller)

Now you have defined your custom draggable/droppable view – what you should do next is to populate all instances (in our first sample project: all country objects) with real data. You'll do it in the **View Controller** – usually in a loop, iterating over collections (arrays, dictionaries or even Core Data):

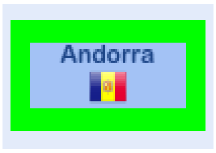
```
for (int i=0; i<countries.count; i++) {
    // first populate the draggable view from framework (which serves as
    // a container for the custom view - see below)
    ArrasoltaDraggableView* view = [ArrasoltaDraggableView new];
    view.index = i; // index must be set - otherwise, the undo functionality won't
        work
    [view setBorderColor:[UIColor redColor]];
    [view setBorderWidth:IS_IPAD ? 2 : 1];

    // now populate the own UIView which serves as draggable/droppable view
    CustomView* customView = [CustomView new];
    // make alternating background colors
    if (i%2==0) {
        customView.backgroundColorOfView = [UIColor colorWithRed:0.64 green:0.76
            blue:0.96 alpha:1.0];
    } else {
        customView.backgroundColorOfView = [UIColor colorWithRed:0.43 green:0.62
            blue:0.92 alpha:1.0];
    }
    customView.labelText = countries[i][0];
    customView.labelColor = [UIColor colorWithRed:0.11 green:0.27 blue:0.53 alpha:
        1.0];
    customView.imageName = countries[i][1];

    [view setContentView:customView];

    // the dictionary must contain Numbers as Key and
    // ArraSoltaDragView objects as value
    dataSourceDictionary[@(i)] = view;
}
```

For each item, you create a new instance of your custom view and set the **public properties** (in this case ***labelText***, ***labelColor*** and ***imageName***). As already mentioned, don't set the subview components (UILabel and UIImageView) directly!

As already described in **5.1 View Controller**, you should populate your data source (for the source collection view) with ***ArrasoltaDraggableView*** rather than with your custom view objects! ***ArrasoltaDraggableView*** is a wrapper class and has two further setters:

1. ***borderColor***,
2. ***borderWidth***.

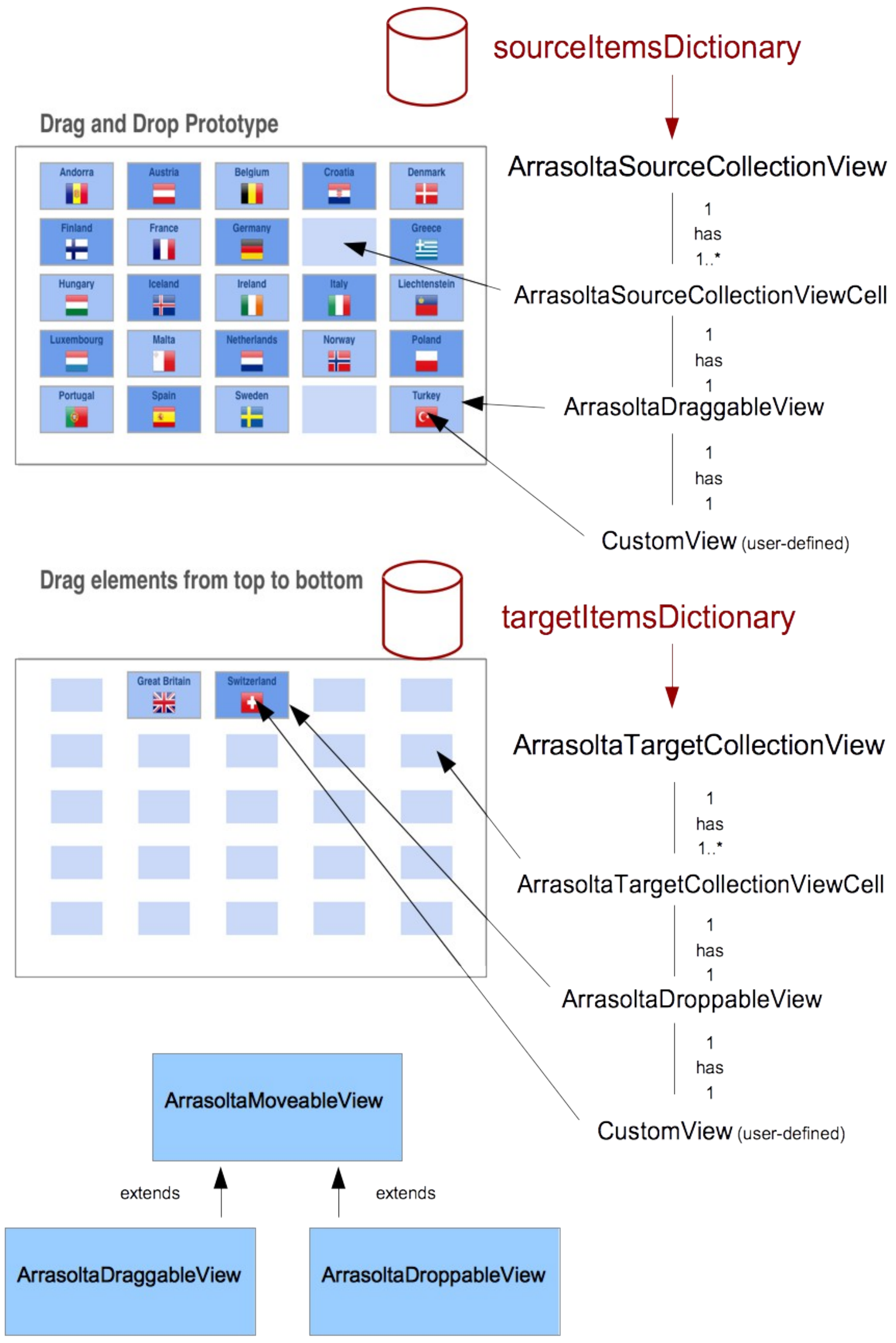
Let's see what happens if you set the borderColor to green and the borderWidth to 10:



You see that the wrapper class ***ArrasoltaDraggableView*** serves as a container for your custom view!

# 6 Architecture of the ArraSolta Framework

The following diagram provides an overview of the architecture and helps you to better understand the implementation details mentioned in the previous chapters:



Let's explain the dependencies between the components shown in the diagram above:

1. Each collection view is filled by data from NSMutableDictionary. Source Collection View is populated at the beginning, Target Collection View dynamically after user interaction – it's the framework which updates both the source, both the target dictionary.
2. We have two types of collection views – ***ArrasoltaSourceCollectionView*** and ***ArrasoltaTargetCollectionView***. They both extend ***ArrasoltaCollectionView*** – which in return extends ***UICollectionView.***
3. We also have two types of collection view cells – ***ArrasoltaSourceCollectionViewCell*** and ***ArrasoltaTargetCollectionViewCell***. They both extend ***ArrasoltaCollectionViewCell*** – which in return extends ***UICollectionViewCell***. Please note that the reuse identifiers of all cells are declared internally within the framework.
4. Your custom view must be wrapped into ***ArrasoltaDraggableView***. As soon as you drop your custom view from the source into the target collection view, the ***ArrasoltaDraggableView*** is automatically converted (by the framework) into ***ArrasoltaDroppableView***. Both classes extend ***ArrasoltaMoveableView*** – which in return extends ***UIView***.

# 7 ArraSolta API

## 7.1 Arrasolta Config

All method signatures concerning framework configuration are available in ***ArrasoltaConfig.h***:

```
@interface ArrasoltaConfig : NSObject


+ (ArrasoltaConfig*) sharedInstance;

@property (NS_NONATOMIC_IOSONLY, getter=getFixedCellSize) CGSize fixedCellSize;
@property (NS_NONATOMIC_IOSONLY, getter=getCellWidthHeightRatio) float
    cellWidthHeightRatio;
@property (NS_NONATOMIC_IOSONLY, getter=getShouldCollectionViewBeCenteredVertically)
    bool shouldCollectionViewBeCenteredVertically;
@property (NS_NONATOMIC_IOSONLY, getter=getShouldCollectionViewFillEntireHeight) bool
    shouldCollectionViewFillEntireHeight;
@property (NS_NONATOMIC_IOSONLY, getter=getMinInteritemSpacing) float
    minInteritemSpacing;
```

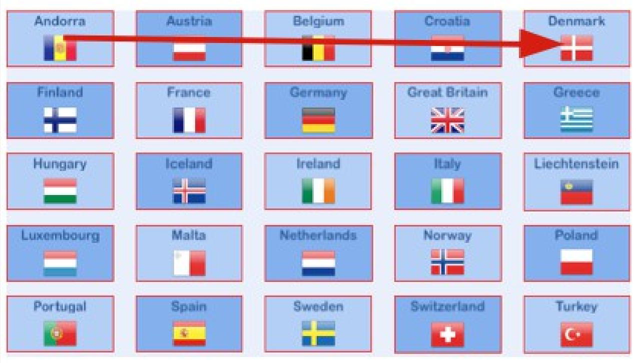
We now explain these methods (getters and setters) more in detail. We recommend that you use the ***SHARED_CONFIG_INSTANCE*** macro.

| Data type | Property as getter/setter | Explanation |
|---|---|---|
| NSMutableDictionary* | *sourceItemsDictionary* | Populates the source collection view with the data source (NSMutableDictionary). **This property must be set!** |
| bool | *sourceItemsConsumable* | Flag whether a cell of the source collection view is consumable. If ***yes***, they can be dragged and dropped only once – if ***not***, source items are continuously replaced as soon as the user drops the item into the target view. |
| int | *numberOfTargetItems* | Number of items of the target collection view. **Must be set if source items are NOT consumable** – thus in such case it makes sense that both collection views have a different numbers of cells. |
| CGSize | *fixedCellSize* | Static size (width and height of each collection view cell) defined by the user. Must be set if cellWidthHeightRatio is empty. **This property (or cellWidthHeightRatio) must be set!** |
| float | *cellWidthHeightRatio* | Aspect ratio of each collection view cell. If set, the framework automatically calculates the best cell size so that all collection view cells fit into the content view without the need of scrolling. Please note that fixedCellSize is stronger: if set, cellWidthHeightRatio won't have an effect! **This property (or fixedCellSize) must be** |

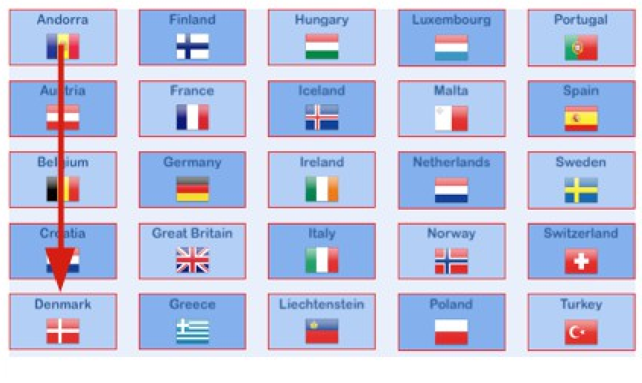
| | | |
|---|---|---|
| | | **set!** |
| bool | *shouldCollectionViewBeCenteredVertically* | In the case that the collection view cells don't fill the content view vertically, they are centred vertically so that the distances to the upper and lower border are the same. |
| bool | *shouldCollectionViewFillEntireHeight* | In the case that the collection view cells don't fill the content view vertically, the line spacings are expanded so that the cells fill completely the content view (no margin at all) |
| float | *minInteritemSpacing* | Analogous to Apple's specification, it describes the minimal horizontal space between the cells. |
| float | *minLineSpacing* | Analogous to Apple's specification, it describes the minimal vertical space between the cells. |
| UIColor* | *backgroundColorSourceView* | Describes the background colour of the source collection view |
| UIColor* | *backgroundColorTargetView* | Describes the background colour of the target collection view |
| UIColor* | *sourcePlaceholderColor* | Describes the background colour of a source collection view cell. The colour is visible as soon as a collection view cell has been dragged away (see consumable cells) |
| UIColor* | *targetPlaceholderColorUntouched* | Describes the background colour of a target collection view cell and makes the placeholder visible to the user. |
| UIColor* | *targetPlaceholderColorTouched* | Describes the background colour of a target collection view cell when the user drags a cell over it. The event-driven change of the colour is useful to the user as he gets the info if he can drop his item into it or not. |
| bool | *shouldSourcePlaceholderDisplayIndex* | Flag whether the placeholder of each source collection view cell should display an index (number) or not. |
| bool | *shouldTargetPlaceholderDisplayIndex* | Flag whether the placeholder of each target collection view cell should display an index (number) or not. |
| bool | *shouldPlaceholderIndexStartFromZero* | Flag whether the index (see above) should start with zero or by one. |
| bool | *placeholderTextColor* | Font colour of the index (see above) |
| float | *placeholderFontSize* | Font size of the index (see above) |
| NSString* | *preferredFontName* | Font name for all collection view items – if not set, the framework uses ***Helvetica-Bold*** has standard font. Please note that you should call the correspondent getter wherever you want to use the same font, as for example in the custom draggable view. |
| NSInteger | *scrollDirection* | Scroll direction of both collection views – |

| | | |
|---|---|---|
| | | possible values are ***horizontal*** and ***vertical*** (=enum representation) |
| bool | *shouldCellOrderBeHorizontal* | Relevant only for horizontal scroll direction: describes if the elements (ascending order) should be placed horizontally or vertically. As default, Apple places them vertically in conjunction with flow layout – see screenshot below! |
| float | *longPressDurationBeforeDragging* | Time (in seconds) for long press gestures in conjunction with dragging an element. If set > 0, the user must first perform a long press gesture before he can drag the elements. If = 0, the user can drag the elements without performing a long press gesture before. Setting the value > 0 makes sense if panning (zooming) is enabled, as he can't drag an item accidentally while panning. |
| bool | *shouldPanningBeEnabled* | Flag whether the collection views are zoomable by panning or not. |
| bool | *shouldPanningBeCoupled* | Flag whether panning should be coupled (synchronized) between both collection views. If yes, panning of one collection view has an impact on the other collection view and vice-versa. |

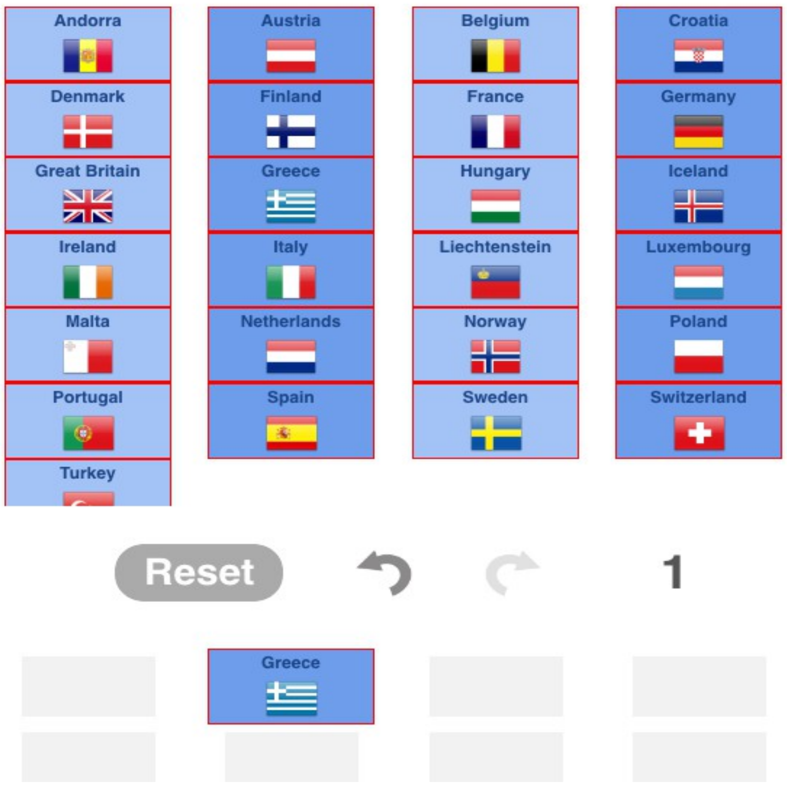
```
[SHARED_CONFIG_INSTANCE setShouldCellOrderBeHorizontal:true]
```



```
[SHARED_CONFIG_INSTANCE setShouldCellOrderBeHorizontal:false]
```



As shown in the table above, only a minimum of properties must be set – most properties are optional and return a default value if not explicitly set – with only a minimum of properties set, you're likely to get something like this:



You see that background colours are missing – the line spacing between the items is zero, the colours of the placeholders are grey – and the scroll direction is set to vertically. Besides, the cells are not consumable: you see "Greece" twice.

Of course you can experiment with the above-mentioned properties in both Xcode sample projects.

**Important:**

***Scrolling synchronization***

Although not part of the API, you should know how to synchronize the scrolling between both collection views – you can use the following code (and modify it) in your UIView containing the collection views:

```
#pragma mark <UIScrollViewDelegate>
// synchronize scrolling
- (void)scrollViewDidScroll:(UIScrollView *)scrollView {

    CGPoint currentOffset = scrollView.contentOffset;
    // if source collection view is scrolled, scroll also the target collection view (but NOT vice-versa!)
    if ([scrollView isKindOfClass:[ArrasoltaSourceCollectionView class]]) {
        self.targetCollectionView.contentOffset = currentOffset;
    } else {
        //self.dragCollectionView.contentOffset = currentOffset;
    }
}
```

## 7.2 Arrasolta Utils

This utility class comes with 12 methods, but you will only need the following ones:

| Return type | Method | Explanation |
|---|---|---|
| ArrasoltaCollectionViewCell* | *getCell:(UICollectionView *)collectionView forIndexPath:(NSIndexPath *)indexPath cellDictionaries:(NSArray*) cellDictionaries* | This method returns a single collection view cell and **must be implemented** inside ***cellForItemAtIndexPath*** - as described in ***5.3.3 UICollectionViewDataSource Protocol*** |
| UIColor* | *getRandomColor* | Helper method – generates a random colour and can be used for test issues (for example generating a collection view with different cell background colours). |
| void | *scrollToLastElement: (UICollectionView*) collectionView ofDictionary: (NSMutableDictionary*) dict* | This helper method is useful if – for example after device rotation – the collection view should scroll to the last element. This method is optional. |

## 7.3 Arrasolta Undo Button Helper

The following methods can be called using the ***SHARED_BUTTON_INSTANCE*** macro. You only need to bind the following GUI elements to the helper class – the bindings between helper class and dictionaries are made internally by the framework. All methods are optional, as the button bar is only an optional feature for the drag and drop functionality.

| Return type | Method | Explanation |
|---|---|---|
| void | *initWithUndoButton: (UIButton*) button* | Binds the undo button to the framework. |
| void | *initWithRedoButton: (UIButton*) button* | Binds the redo button to the framework. |
| void | *initWithResetButton: (UIButton*) button* | Binds the reset button to the framework. With the aid of the reset button, you can clear the target collection view and (if cells are consumable) move them back to the source collection view. |
| void | *initWithInfoLabel: (UILabel*) label* | Binds the counter label to the framework. The counter is useful for tracking all user interactions, such as inserting, deleting, moving items. |

# 8 Cheat Sheet – Implementation at a glance

Implementing ArraSolta is easy – here comes a guideline about the most important steps – for details, please study the previous chapters:

1. Copy the ***ArraSolta.framework*** file into your project directory. You can place it into a subdirectory (=Framework) also.

2. Tell Xcode where it can find the framework.

3. Import the master header file ***ArrasoltaAPI.h*** into all classes dealing with drag and drop (or into a centralized PCH file). As it deals with a framework rather than with a class file, you should use brackets: ***#import <ArraSolta/ArrasoltaAPI.h>.***

4. Make all framework-specific configurations (see API) in the View Controller – as for example the cell width-height ratio. Use the macro ***SHARED_CONFIG_INSTANCE***.

5. Develop your custom view class (draggable view) and watch out that subview components are not declared as public. Instead, generate public getters/setters for the subview's properties, such as text colour, image name, etc.

   - Your custom class must extend ***ArrasoltaCustomView***.
   - Don't forget the following line of code in the init method:
     ***self.concreteClassName = NSStringFromClass([self class])***.

6. Go back to the View Controller and create instances of your custom view.
   - Use the public setters in order to populate your instances with data.
   - Wrap the instances of your custom view into ***ArrasoltaDraggableView.***
   - Put them into your ***NSMutableDictionary*** (data source for source collection view).

7. Tell the framework about the new data source – in the View Controller, set something like
   ***[SHARED_CONFIG_INSTANCE setSourceItemsDictionary:dataSourceDictionary];***

8. Develop an **UIView** containing both collection views. If it deals with a new Xcode project, don't forget to tell the View Controller about the view it manages – you can write something like this:
   ***self.view = [MainView new];***

9. In the init method of your ***UIView***,
   - define the dictionaries (they must be members of your UIView class) and
   - the number of items of both collection views.
   - Instantiate the collection views using the framework-specific constructors – ***ArrasoltaSourceCollectionView*** and ***ArrasoltaTargetCollectionView***.
   - You should also instantiate the drag and drop helper class:
     ***[[ArrasoltaDragDropHelper sharedInstance] initWithView: ...... ];***

10. In your ***UIView***, implement the methods of ***UICollectionViewDataSource*** protocol:
    - ***numberOfItemsInSection*** and
    - ***cellForItemAtIndexPath*** → return ***[ArrasoltaUtils getCell: .... ];***

11. You won't need to implement the methods of ***UICollectionViewDelegateFlowLayout***:
    - ***sizeForItemAtIndexPath*** is handled internally within the framework!

12. In your ***UIView***, use the **NSNotificationCenter** for observing device rotations.
    You should:
    - perform a ***reload*** of both ***collection views*** after device rotation, as well as
    - an ***update*** of the ***view constraints*** in the case of auto layout.
    - For scrolling issues, you can use the utility method
      ***[ArrasoltaUtils scrollToLastElement … ];***