APPCOREKIT

Bindings for iOS

https://twitter.com/appcorekit

https://github.com/wherecloud/AppCoreKit

http://cocoapods.org/?q=appcorekit
pod AppCoreKit/Binding

Motivations

Apple provides some awesome technology implementing different patterns for getting notified of changes in object contents or UI control state. This includes:

Key-Value Observing
https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/KeyValueObserving/KeyValueObserving.html

Notifications
https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSNotificationCenter_Class/Reference/Reference.html

Target-Action
https://developer.apple.com/library/ios/documentation/general/conceptual/Devpedia-CocoaApp/TargetAction.html

Using these technologies involves mastering different APIs that all serve the same purpose: Getting notified for changes.

Several of the above patterns require the developer to manage registration or unregistration of observer objects and to understand precisely the object lifecycle, to avoid runtime exceptions and zombie/deallocated instances.

By implementing the binding APIs in the AppCoreKit we wanted to establish a unified way to work with all these notification patterns and help managing the life cycle of the connections between your models, views and view controllers without needing to explicitly register and unregister any single binding all the time. We also wanted to provide block-based implementations for all these kind of connections.

https://developer.apple.com/library/ios/DOCUMENTATION/Cocoa/Conceptual/Blocks/Articles/00_Introduction.html

As bindings are mainly used for synchronizing UI elements with model changes and models can be mutated in different threads (Network for example), we took care of managing multithreading properly by adding options specifying whether the binding's callback must be executed in the UI thread or in the thread responsible or triggering the notification.

Bindings allow the developer to connect or get notified for modifications to one or several model objects, and we wanted to provide appropriate safety precautions if ever one of these objects gets released while it has bindings. We've been able to achieve this using the CKWeakRef mechanism, which serves as a safe guard in case bindings are not used correctly by the developer.

Finally, we also wanted to automatically convert values from one type to another when connecting models to UI component properties which has been achieved by using the AppCoreKit runtime and transformation APIs.

The Bindings API

The bindings API encapsulates all these notification mechanism in a single API.

NSObject bindings

NSObject bindings allow the developer to get notified for changes in an object's property. This is based on the KVO technology and these bindings manage the registration/unregistration of observers automatically. You can either bind an object's property to another object's property, or execute a block when the property gets set.

As they are based on KVO, these bindings are subject to the same limitations. If you want a binding to work properly and get notified for changes, this means you have to declare and use properties correctly and understand the differences between instance variables and properties.

When you declare a property in your `@interface` combined to `@synthesize` the compiler will automatically generate a setter/getter for you depending on your properties' attributes or if you define setter/getter by yourself. The compiler automatically adds the calls necessary for the KVO notification system to work properly:

```
willChangeValueForKey:
didChangeValueForKey:
```

When modifying the content of your objects, if you want the KVO to trigger the notification, you have to set the property and not the instance variable. Using self.myProperty = something considerably differs from _myProperty = something because the first call will invoke the setMyProperty: method which takes the properties' attributes (retain, assign, ...) and KVO notification into account.

```
@interface NSObject (CKBindings)

- (void)bind:(NSString *)keyPath toObject:(id)object withKeyPath:
(NSString *)keyPath;

- (void)bind:(NSString *)keyPath withBlock:(void (^)(id value))block;

- (void)bind:(NSString *)keyPath executeBlockImmediatly:(BOOL)execute
withBlock:(void (^)(id value))block;

- (void)bind:(NSString *)keyPath target:(id)target action:
(SEL)selector;
```

```
- (void)bindPropertyChangeWithBlock:(void (^)(NSString* propertyName,
id value))block;

@end
```

UIControl Bindings

UIControl's bindings allows to get notified for user actions such as touches or value changes in the case of textField or sliders for example and this manage the registration/unregistration of targets automatically.

```
@interface UIControl (CKBindings)

- (void)bindEvent:(UIControlEvents)controlEvents withBlock:(void (^)
())block;

- (void)bindEvent:(UIControlEvents)controlEvents
executeBlockImmediatly:(BOOL)execute withBlock:(void (^)())block;

- (void)bindEvent:(UIControlEvents)controlEvents target:(id)target
action:(SEL)selector;

@end
```

NSNotificationCenter Bindings

NSNotificationCenter bindings allows to get notified for any notification sent by the notification center and this manage the registration/unregistration of observers automatically.

```
@interface NSNotificationCenter (CKBindings)

- (void)bindNotificationName:(NSString *)notification object:
(id)notificationSender withBlock:(void (^)(NSNotification
*notification))block;

- (void)bindNotificationName:(NSString *)notification withBlock:(void
(^)(NSNotification *notification))block;

- (void)bindNotificationName:(NSString *)notification object:
(id)notificationSender target:(id)target action:(SEL)selector;

- (void)bindNotificationName:(NSString *)notification target:
(id)target action:(SEL)selector;
```

```
+ (void)bindNotificationName:(NSString *)notification object:
(id)notificationSender withBlock:(void (^)(NSNotification
*notification))block;

+ (void)bindNotificationName:(NSString *)notification withBlock:(void
(^)(NSNotification *notification))block;

+ (void)bindNotificationName:(NSString *)notification object:
(id)notificationSender target:(id)target action:(SEL)selector;

+ (void)bindNotificationName:(NSString *)notification target:
(id)target action:(SEL)selector;

@end
```

CKCollection Bindings

CKCollection is a wrapper on top of NSArray provided in the AppCoreKit. It has several advantages compared to NSArray such as the capacity to get paired with a feed source to abstract the source of its content as well as providing APIs to fetch the content from this feed source on demand. The major parts of our ready to use collection view controllers are able to manage CKCollection object and listen for changes to sync the UI automatically.

CKCollection bindings allows to get notified for insertion or deletion of objects in the collection.

```
typedef NS_ENUM(NSInteger, CKCollectionBindingEvents){
    CKCollectionBindingEventInsertion   = 1 << 0,
    CKCollectionBindingEventRemoval     = 1 << 1,
    CKCollectionBindingEventAll = CKCollectionBindingEventInsertion |
CKCollectionBindingEventRemoval
};

@interface CKCollection (CKBindings)

- (void)bindEvent:(CKCollectionBindingEvents)events withBlock:(void(^)
(CKCollectionBindingEvents event, NSArray* objects, NSIndexSet*
indexes))block;

- (void)bindEvent:(CKCollectionBindingEvents)events
executeBlockImmediatly:(BOOL)executeBlockImmediatly withBlock:(void(^)
(CKCollectionBindingEvents event, NSArray* objects, NSIndexSet*
indexes))block;

@end
```

Managing Bindings Life Cycle

When creating a binding using one of the previous APIs, some observers will get registered to the appropriate Apple technology. As such, each binding still has to get unregistered or exceptions or logs can occur.

You are already familiar with this kind of pattern if you already used core animation to group several views animations in one single context. This allows to clear all the animations in this context with one single call.

Bindings Context is a very similar idea allowing to instantiate and group bindings in a way that you can clear all the connections at once, which will unregister all the corresponding observers from the responsible stack of apple's technology and delete these bindings. Bindings MUST be instantiated inside a context or the framework will assert because it will have no clues when to remove it.

1. Open the binding context with the options
2. Instantiate your bindings
3. End the Binding Context

When opening a context, we provide several options to manage whether this context must be cleared prior to populating it with new bindings, or instead keep the existing ones. Also, we specify if the callbacks of the these bindings must be executed on the UI thread or in the thread responsible for the notification.
By default, bindings callbacks will be executed on the UI thread because most of the time, it is dedicated to update or get notified for changes in the UI.

Contexts can be nested which means, if you open a new context 'B' while a context 'A' is opened, after ending 'B', 'A' is the current context for bindings until 'A' gets terminated.

A context is identified by an Object of any kind. Most of the time, it will be the instance where the bindings are taking place. But in some cases, it is useful to open several context at once, to manage the duration of several sets of bindings independently. In this case, we generally build a string identifying the purpose and the instance for the context as follows:

```
NSString* bindingContext =
          [NSString stringWithFormat:@"MyPurpose_<%p>",self];

[NSObject beginBindingsContext:bindingContext
          policy:CKBindingsContextPolicyRemovePreviousBindings];
```

Most of the time, we'll use the Object's instance APIs as follows:

```
[self beginBindingsContextByRemovingPreviousBindings];
```

The binding context management API:

```
@interface NSObject(CKBindingContext)

+ (void)beginBindingsContext:(id)context;

+ (void)beginBindingsContext:(id)context policy:
(CKBindingsContextPolicy)policy;

+ (void)beginBindingsContext:(id)context options:
(CKBindingsContextOptions)options;

+ (void)beginBindingsContext:(id)context policy:
(CKBindingsContextPolicy)policy options:
(CKBindingsContextOptions)options;

+ (void)endBindingsContext;

+ (void)removeAllBindingsForContext:(id)context;


- (void)beginBindingsContextByKeepingPreviousBindings;

- (void)beginBindingsContextByRemovingPreviousBindings;

- (void)beginBindingsContextByKeepingPreviousBindingsWithOptions:
(CKBindingsContextOptions)options;

- (void)beginBindingsContextByRemovingPreviousBindingsWithOptions:
(CKBindingsContextOptions)options;

- (void)endBindingsContext;

- (void)clearBindingsContext;

@end
```

In the AppCoreKit, most of the classes where we intensively use bindings automatically clear their bindings context when they get deallocated as a safe guard, freeing the developer from this concern. This is the case of:

CKViewController
CKUITableViewCell
CKCollectionCellController
CKFormSectionBase
CKPopoverController

How to use bindings to synchronize my models and my UI?

Here are some samples of how to use bindings to ensure your UI and models stay in sync at any time even if your model can be mutated by several different components of your application. It is really important to understand the fact that UI should act on the models and should react to models' changes. The state of a UI component MUST not be used to set the model or it can lead to synchronization problems. This idea is well illustrated in the following button sample.

Sample: Button

Lets illustrate a buttons that must be selected/unselected in sync with a model's BOOL property.

```
@interface MyModel : NSObject
@property (nonatomic, assign) BOOL myBool;
@end


@interface MyViewController : UIViewController
@end

@implementation MyViewController

- (void)viewWillAppear:(BOOL)animated{
    [super viewWillAppear:animated];

    [self beginBindingsContextByRemovingPreviousBindings];
    [self setupMyButton];
    [self endBindingsContext];
}

- (void)setupMyButton{
    //Get your model and your button as needed.
    MyModel* model     = [MyModel sharedInstance];
    UIButton* myButton = (UIButton*)[self.view viewWithTag:10];

    //This ensure the button selected state stay in sync whenever the model's
    //property changes and when setupMyButton gets called.
    [model bind:@"myBool" executeBlockImmediatly:YES withBlock:^(id value) {
        myButton.selected = model.myBool;
    }];

    //This just toggles the model's value
    [myButton bindEvent:UIControlEventTouchUpInside withBlock:^{
        model.myBool = !model.myBool;
    }];
}

@end
```

In this example, we see that the button does not toggle its own state but acts on the models' property. In the same time, the view controller is bound to the models' property changes and sets the button state accordingly.
By this way we ensure that the state of the button will always reflect the models' property value for the life time of this controller wherever the changes occur!

Sample: Textfield

```
@interface MyModel : NSObject
@property (nonatomic, assign) NSString* myString;
@end

@interface MyViewController : UIViewController
@end

@implementation MyViewController

- (void)viewWillAppear:(BOOL)animated{
    [super viewWillAppear:animated];

    [self beginBindingsContextByRemovingPreviousBindings];
    [self setupMyTextField];
    [self endBindingsContext];
}

- (void)setupMyTextField{
    //Get your model and your textfield as needed.
    MyModel* model         = [MyModel sharedInstance];
    UITextField* textField = (UITextField*)[self.view viewWithTag:10];

    [model bind:@"myString" executeBlockImmediatly:YES withBlock:^(id value){
        //Ensure we don't have a recursive loop
        if(![textField.text isEqualToString:model.myString]){
            textField.text = model.myString;
        }
    }];

    //This sets the model value when textfield text changes.
    [textField bindEvent:UIControlEventEditingChanged withBlock:^{
        model.myString = textField.text;
    }];
}

@end
```

The same process is valid for sliders, or any UIControl managing a value.

These samples are pretty simple but as soon as you'll have several views or controls needing updates as a reaction of one or several property changes, it will considerably be helpful as all the views can be managed independently from each other. You're now free to change your design and just move or remove the piece of code that concerns this view without untangling a can of worms.

Using Blocks = WARNING!

People like to think that ARC is magic and that it ensures the integrity of the memory management in all cases. This is not true especially when working using blocks. Using blocks can easily lead to retain cycles and memory leaks if they are not used carefully. However, these pitfalls can be avoided with some simple rules that quickly become natural.

Here is a sample that will lead to a memory leak:

```
@implementation MyViewController

- (void)viewWillAppear:(BOOL)animated{
    [super viewWillAppear:animated];

    [self beginBindingsContextByRemovingPreviousBindings];
    [self setupSomeBindings];
    [self endBindingsContext];
}

- (void)setupSomeBindings{
    //Get your model and your textfield as needed.
    MyModel* model         = [MyModel sharedInstance];

    [model bind:@"myString" executeBlockImmediatly:YES withBlock:^(id value){
        self.view.backgroundColor = [UIColor redColor];
    }];
}

@end
```

What happens here is that we are using `self` inside the bindings' block. This block will therefore retain self until it gets deallocated. In the same time, this block will get deallocated when the binding context will get cleared and the binding context is `self`. As the context is supposed to be cleared when `self` is deallocated, this will never happen because self is retained by the binding that depends on `self` life cycle!

Here is a reflex to apply anytime you are working with blocks and you know some instances you are referencing in blocks MUST NOT be retained.

```
__unsafe_unretained MyViewController* bself = self;
[model bind:@"myString" executeBlockImmediatly:YES withBlock:^(id value) {
     bself.view.backgroundColor = [UIColor redColor];
}];
```

By assigning self in an `__unsafe_unretained` variable, when using bself into the block, `self` will not get retained but bself will point to the same instance as `self` (which obviously is what we want here).

Finally, do not uses `NSAssert` inside blocks as it is replaced by code using self.